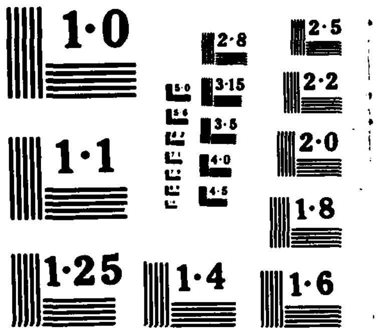
AD-A187 031   FX-87 REFERENCE MANUAL EDITION 1.0(U) MASSACHUSETTS INST   1/2
OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE
D K GIFFORD ET AL   SEP 87 MIT/LCS/TR-407

UNCLASSIFIED   N00014-83-K-0125   F/G 12/5   NL


1·0
2·8
2·5
3·15
2·2
5·0
3·5
1·1
2·0
4·0
4·5
1·8
1·25
1·4
1·6

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

A187031 (4)

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution is unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>MIT/LCS/TR-407 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>DARPA/DOD N00014-83-K-0125 |
| 6a. NAME OF PERFORMING ORGANIZATION<br>MIT Lab for Computer Science<br>6b. OFFICE SYMBOL (If applicable) | 7a. NAME OF MONITORING ORGANIZATION<br>Office of Naval Research/Dept. of Navy |
| 6c. ADDRESS (City, State, and ZIP Code)<br>545 Technology Square<br>Cambridge, MA 02139 | 7b. ADDRESS (City, State, and ZIP Code)<br>Information Systems Program<br>Arlington, VA 22217 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>DARPA/DOD<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State, and ZIP Code)<br>1400 Wilson Blvd.<br>Arlington, VA 22217 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

DTIC S ELECTE NOV 1 8 1987 A

11. TITLE (Include Security Classification)
FX-87 REFERENCE MANUAL

12. PERSONAL AUTHOR(S)
Gifford, David K.; Jouvelot, Pierre; Lucassen, John M.; and Sheldon, Mark A.

| | | | |
|---|---|---|---|
| 13a. TYPE OF REPORT<br>Technical | 13b. TIME COVERED<br>FROM ____ TO ____ | 14. DATE OF REPORT (Year, Month, Day)<br>September 1987 | 15. PAGE COUNT<br>148 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | |
|---|---|---|
| FIELD | GROUP | SUB-GROUP |
| | | |
| | | |

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Programming Languages, types, effects, regions, polymorphism, static checking, optimization, parallel programming

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The FX programming language is designed to support the parallel implementation of applications that perform both symbolic and scientific computations. Unlike previous languages, FX uses an effect system to discover expression scheduling constraints. The effect system is part of a kinded type system with three base kinds: types, which describe the value of an expression; effects, which describe the side-effects that an expression may have; and regions, which describe the areas of the store in which side-effects may occur. Types, effects, and regions are collectively called descriptions.

FX expressions can be abstracted over any kind of description. This permits type, effect, and region polymorphism. Unobservable side-effects are masked by the effect sytem; an effect soundness property guarantees that the effects computed statically by the effect system are a conservative approximation of the actual side-effects that a given expression may have.

Effect polymorphism and effect masking make the FX effect system substantially more

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>[✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>Judy Little | 22b. TELEPHONE (Include Area Code) (617) 253-5894 / 22c. OFFICE SYMBOL |

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

*U.S. Government Printing Office: 1985-507-047

7 1 03

19. powerful than previous approaches to side-effect analysis.

MIT/LCS/TR-407

# *FX-87* Reference Manual

David K. Gifford
Pierre Jouvelot
John M. Lucassen
Mark A. Sheldon

September 1987
Edition 1.0

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

# Abstract

The *FX* programming language is designed to support the parallel implementation of applications that perform both symbolic and scientific computations. Unlike previous languages, *FX* uses an *effect system* to discover expression scheduling constraints. The effect system is part of a *kinded* type system with three base kinds: *types,* which describe the value of an expression; *effects,* which describe the side-effects that an expression may have; and *regions,* which describe the areas of the store in which side-effects may occur. Types, effects, and regions are collectively called *descriptions.*

*FX* expressions can be abstracted over any kind of description. This permits type, effect, and region polymorphism. Unobservable side-effects are *masked* by the effect system; an effect soundness property guarantees that the effects computed statically by the effect system are a conservative approximation of the actual side-effects that a given expression may have.

Effect polymorphism and effect masking make the *FX* effect system substantially more powerful than previous approaches to side-effect analysis.

**Keywords :**

Programming Languages, Types, Effects, Regions, Polymorphism, Static Checking, Optimization, Parallel Programming [illegible]

# Contents

# Preface

The *FX* programming language is designed to support the parallel implementation of applications that perform both symbolic and scientific computations. Unlike previous languages, *FX* supports both functional and imperative parallel programming by a static checking system based upon the notion of *effects*. Just as a type describes what an expression computes, an effect describe how an expression computes. An effect is a static description of the observable side-effects that an expression may have when it is evaluated.

When a programmer uses *FX*, opportunities for parallel evaluation are automatically identified by the *FX effect system*. The effect system assigns an effect to each expression in a program. Since the effects of every *FX* expression are statically known, effect information can be used to schedule a program for parallel evaluation while retaining sequential semantics. If two expressions do not have interfering effects, then a compiler can schedule them to run in parallel subject to dataflow constraints.

The effect classifications used by *FX* include `read` effects, `write` effects, and `alloc` (for allocate) effects. Each effect is subscripted by the region of the store to which it applies. Compound effects are built from unions of simple effects, and thus effects form a lattice. The bottom of the effect lattice is the effect pure, which is used to describe an expression that has no effect at all.

The *FX* effect system uses *effect masking* to erase unobservable side-effects from the effect of an expression. Effect masking allows expressions that have local side-effects to be scheduled to run in parallel with one another. In addition, the same static analysis that is used for effect masking permits local storage to be stack allocated, thus saving the overhead of dynamic garbage collection.

The *FX* static checking framework is based on a hierarchical kinded type system which includes kinds, universal polymorphism, higher order types, and recursive types. The static checking system is based upon three kinds of *descriptions*: types, which describe the values expressions compute; effects, which describe the side-effects of expressions; and regions, which describe where effects occur. An expression may be polymorphic in any of the three kinds. Thus the type of a subroutine may depend on the effect parameters passed to it. Effect and region polymorphism permit the effect system to provide tight effect bounds on higher-order functionals in a natural and simple manner.

We have found that an effect system is useful to programmers, compiler writers, and language designers in the following respects:

- An effect system lets the *programmer* specify the side-effect properties of program modules in a way that is machine-verifiable. The resulting effect specifications are a natural extension of the type specifications found in conventional programming languages. We believe that the use of effect specifications has the potential to improve the design and maintenance of imperative programs.

- An effect system lets the *compiler* identify optimization opportunities which are hard to detect in a conventional higher-order imperative programming language. We have focused our research on three classes of optimizations: code motion (including eager, lazy, and parallel evaluation); common subexpression elimination (including memoization); and dead code elimination. We believe that the ability to perform these optimizations effectively in the presence of side-effects represents a step towards integrating functional and imperative programming for the purpose of massively parallel programming.

- An effect system lets the *language designer* express and enforce side-effect constraints in the language definition. In *FX*, for example, the body of a polymorphic expression must not have any side-effects. This restriction not only simplifies the type system by making effect specifications on polymorphic types unnecessary, but also makes this the first language known to us that permits an efficient implementation of fully orthogonal polymorphism, in which any expression can be abstracted over any type and all polymorphic values are first-class, in the presence of side-effects.

## Organization of the *FX* Reference Manual

The *FX* manual is organized into eight major parts:

- An overview of *FX* conventions. These conventions include the meta-notation used throughout the manual, how dynamic and static errors are documented, and the reserved keywords of *FX*.

- The *FX* Primer, a short introduction to the use of *FX* with examples (Chapter 1).

- The *FX* Kernel, which includes essential *FX* constructs and the *FX* type and effect system (Chapter 2). The Kernel forms the core of the language from the point of view of both the *FX* application programmer and the *FX* language implementor.

- Standard *FX* types and operations on refs, booleans, integers, floats, vectors, lists, oneofs, records, and so forth (Chapter 3).

- *FX* Syntactic Sugar for frequently used Kernel constructs (Chapter 4). Sugar forms (such as `let`) do not add semantic power to the language because they can be described directly in terms of more primitive Kernel constructs.

- The *FX* Environment for programming (Chapter 5). The environment includes I/O facilities, top-level definitions, and facilities for developing large *FX* programs.

- The BNF syntax of *FX* (Appendix A). The syntax describes all of the special forms in value, description, and kind expressions.

- The semantics for the *FX* Kernel (Appendix B). The semantics is used to prove the soundness of the parallel optimizations which are permitted by the type and effect system.

This report corresponds to *FX-87*. An *FX-87* interpreter written in Scheme can be obtained by sending an electronic mail request to `gifford@xx.lcs.mit.edu`. Subsequent versions of *FX* will include such extensions as separate compilation, exception handling, type inference, and explicit concurrency.

The *FX-87* programming language was developed by the Programming Systems Research Group at MIT. In addition to the authors, Mike Blair, Mark Day, Jonathan Rees and James O'Toole made contributions to the design of *FX-87*. James O'Toole designed and documented the facilities for implicit projection. Kendra Tanacea provided helpful comments on drafts of this report. The design of *FX* was strongly influenced by Scheme, especially in the choice of standard types and operations.

Your comments on this report are welcome.

## References

David K. Gifford and John M. Lucassen. *Integrating Functional and Imperative Programming*. Proc. 1986 ACM Conference on LISP and Functional Programming, August 4-6, 1986, Cambridge MA, pp. 28-38.

John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD Dissertation. MIT/LCS/TR-408, 1987.

John M. Lucassen and David K. Gifford. *Polymorphic Effect Systems.*, Proc. 15th Annual ACM Conference on Principles of Programming Languages, January 1988.

Jonathan Rees, *et al.* *Revised$^3$ Report on the Algorithmic Language Scheme*. AI Memo 848a. MIT, Artificial Intelligence Laboratory, September 1986.

# Conventions

This chapter presents the conventions that are used throughout this manual and introduces *FX* syntactic notation, how dynamic and static errors are documented, *FX* syntactic classes, and *FX* reserved identifiers.

## Syntactic Notation

This manual adheres to the following conventions:

*FX* program text is written in `teletype font`. Program text is comprised of identifiers, literals, and delimiters.

Meta-expressions, which are names for syntactic *classes* of expressions, are written in *italic font*. A programmer may replace any meta-expression by a compatible *FX* expression. Meta-expressions are distinguished by their suffix. For example, meta-variables end with *var* and meta-expressions end with *exp*.

In the following example specification, `if` is a reserved identifier and the $exp_i$ denote any valid *FX* expression:

### Example:

(`if` $exp_0$ $exp_1$ $exp_2$)

Certain *FX* language constructs are specified to take a variable number of parameters. [*exp*] denotes an optional expression. A possibly empty sequence of $n$ expressions is noted $exp_1 \ldots exp_n$. If the name of the upper bound on subscripts is not used, we write the shorter: $exp_1 \ldots$. If there is at least one expression in the sequence (*i.e.*, $n \geq 1$), we use $exp_1\ exp_2 \ldots exp_n$. We usually denote by $exp_i$ (or any other subscripted *exp*) any expression which belongs to any of these sequences.

When a given *FX* construct cannot be kinded (for description expressions) or typed (for an ordinary expression) using the standard *FX* notations, it is described in a special format (double-barred page) in which explanations are given on its syntax, kind, type, effect and/or semantics. One or more examples of its usage are provided.

## Static and Dynamic Errors

*Static errors* are detected by *FX* when a program is type and effect checked. All syntax, type, and effect errors are detected statically and reported. The sentence "*x* must be *y*" indicates that "it is a static error if *x* is not *y*".

*Dynamic errors* may be detected by *FX* when a program is run. The phrase "a dynamic error is signalled" indicates that *FX* implementations must report the corresponding dynamic error and terminate the execution of the program. The phrase "it is a dynamic error" indicates that *FX* implementations do not have to detect or report the corresponding dynamic error. The meaning of a program that contains an unreported dynamic error is undefined.

## Definitions

Here we describe the basic lexical entities used in the *FX* programming language:

- A *digit* is one of `0` ... `9`.
- A *letter* is one of `a` ... `z` or `A` ... `Z`.
- The set of extended alphabetic characters must include: `*`, `/`, `<`, `=`, `>`, `!`, `?`, `:`, `$`, `%`, `_`, `&`, `~`, `^`
- *White space* is a blank space, a newline character, a tab character, or a newpage character.
- A *character* is a digit, a letter, an extended alphabetic character, `+`, `-`, a white space or backspace character.
- A *delimiter* is a white space, a left parenthesis or a right parenthesis.
- A *token* is a sequence of characters that is separated by delimiters.
- A *literal* is either a number, or a token that begins with `'` or `#`, or a sequence of characters enclosed in double quotes `"`, or an empty set of parenthesis `()`.

- A *number* is a token made of a non empty sequence of digits, possibly including base information, a decimal point, and a sign. (see Chapter 3).

- An *identifier* is a token beginning with a letter or extended alphabetic character and made of a non-empty sequence of letters, digits, extended alphabetic characters, and the characters + and -. Note that + and - by themselves are also identifiers.

*FX* reserves the following identifiers. Reserved identifiers must not be bound, redefined, or used as tags for records and oneofs.

| | | | | |
|---|---|---|---|---|
| `alloc` | `do` | `oneof` | `read` | `type` |
| `and` | `effect` | `or` | `record` | `unit` |
| `begin` | `else` | `pairof` | `recordof` | `uniqueof` |
| `bool` | `if` | `pdefine` | `record-set!` | `vectorof` |
| `compile` | `lambda` | `plambda` | `ref` | `vlambda` |
| `cond` | `let*` | `plet*` | `region` | `void` |
| `define` | `let` | `plet` | `runion` | `vsubr` |
| `delay` | `letrec` | `pletrec` | `select` | `write` |
| `dfunc` | `load` | `poly` | `set!` | |
| `dlambda` | `maxeff` | `proj` | `string` | |
| `dlet*` | `null` | `promise` | `subr` | |
| `dlet` | `one` | `pure` | `tagcase` | |
| `dletrec` | `one-set!` | `quote` | `the` | |

Comments in *FX* are sequences of characters beginning with a ";" and ending with the end of line on which ";" is located. They are discarded by *FX* and treated as a single whitespace.

# Chapter 1

# The FX Primer

This primer is designed to introduce *FX* to you by way of a few representative programs – even before you have read the rest of this manual. We hope that this primer will give you the flavor of programming in *FX*. We also hope to suggest how larger *FX* programs can be created by composing elementary *FX* expressions together.

## Getting Started

In order to use *FX*, start by invoking the *FX* interpreter. *FX* will prompt you with a greeting similar to this one:

```
FX 1.0 Interpreter of June 30, 1987

FX =>
```

*FX* is now ready to listen to you. Once you have typed a complete expression and pressed carriage-return, *FX* will evaluate your expression and output its value. For example:

```
FX 1.0 Interpreter of June 30, 1987

FX => 1
1 : int ! pure
FX => (+ 1 (* 2 3))
7 : int ! pure
FX => (exp 1.0)
2.7182818 : float ! pure
```

```
FX => (> 1 2)
#f  :  bool  !  pure
```

*FX* prints three result components for each expression that you type. The first component is the value of the expression, the component after the " : " is the *type* of the expression, and the component after the " ! " is the *effect* of the expression. The type of an expression describes its value, while the effect of an expression describes its side-effects. Thus, types describe "what" while effects describe "how". All of the above examples had effect *pure*, indicating that no side-effects have been performed by these computations.

The names +, *, exp, and > are simply variables that are bound in the global environment to primitive subroutines. You can bind new variables to values with `define`:

```
FX => (define x 2)
x = 2  :  int  !  pure
FX => x
2  :  int  !  pure
FX => >
<subr>  :  (subr pure (int int) bool)  !  pure
```

In this example, the `(define x 2)` form introduces a new variable in the global environment called x that is initially bound to the value 2. By typing the expression x we confirm that x is bound to the value 2. By typing the expression > we see that > is bound to a subroutine (which cannot be printed, hence the `<subr>`) that takes two integers as input and returns a boolean. The type of > includes the effect that > will have when it is applied. This effect is called the *latent effect* of the subroutine; the latent effect of > is *pure*.

## Using Regions and Effects

Every data structure in *FX* is in some *region*. Regions are useful because they enable *FX* to perform automatic garbage collection and to evaluate more expressions in parallel than would otherwise be feasible. Region constants are easily recognizable because they always start with a `@` character. When you do not specify the region of a data structure, the region `@=` is generally used. Data structures that are in `@=` cannot be mutated. You can choose the region of a data structure by using the `proj` expression to indicate your region choice. For example:

```
FX => (cons 1 2)
(1 . 2) : (pairof int int @=) ! pure
FX => (car (cons 1 2))
1 : int ! pure
FX => (define y ((proj cons @green) 1 2))
y = (1 . 2) : (pairof int int @green) ! (alloc @green)
FX => (car y)
1 : int ! (read @green)
FX => (set-car! y 2)
#u : unit ! (write @green)
FX => (car y)
2 : int ! (read @green)
```

As shown in this example, effects include region specifications. The effect of allocating and initializing a `@green` pair is `(alloc @green)`, the effect of reading a `@green` pair is `(read @green)`, and the effect of writing a `@green` pair is `(write @green)`. Effects on the region `@=` can be ignored because data structures in `@=` may not be mutated; we say that data structures in `@=` are *immutable*.

So far we have discussed values, types, regions, and effects. With these preliminaries out of the way, we can now consider our first example program – a recursive implementation of the Fibonacci function:

```
FX => (define (fib (n int))
        (the pure int
             (cond ((<= n 0) 0)
                   ((= n 1) 1)
                   (else (+ (fib (- n 1)) (fib (- n 2)))))))
fib = <subr> : (subr pure (int) int) ! pure
FX => (fib 5)
5 : int ! pure
```

The `fib` subroutine takes a single argument called `n` of type `int`. The `the` form is a declaration that the body has no side effects – it is pure – and that `fib` returns an integer.

The Fibonacci function can also be programmed with an iterative implementation:

```
FX => (define (iter-fib (n int))
        (do ((result 0 (+ result result-1))
             (result-1 1 result)
             (counter n (- counter 1)))
            ((<= counter 0) result)))
iter-fib = <subr> : (subr pure (int) int) ! pure
FX => (iter-fib 5)
5 : int ! pure
```

Each clause of the do expression provides an initial value for a loop variable and an expression for updating the variable on each loop iteration. When `counter` is equal to or less than 0, `result` is returned.

*FX* gets a considerable amount of its power from the way that subroutines can be used. In *FX*, subroutines can be stored in data structures, passed to other subroutines as arguments, and returned as the results of subroutines. For example, the following subroutine `compose` composes two integer subroutines:

```
FX => (define (compose (f (subr pure (int) int))
                       (g (subr pure (int) int)))
        (lambda ((x int)) (f (g x))))
compose = <subr> : (subr pure ((subr pure (int) int)
                               (subr pure (int) int))
                              (subr pure (int) int))
                   ! pure
FX => ((compose fib (lambda ((x int)) (+ x 1))) 4)
5 : int ! pure
```

## Polymorphism Permits Subroutines to Work for Many Types and Effects

We can generalize `compose` so that it can work for subroutines of any type by passing the input and output type of the subroutines as a special kind of argument:

```
FX => (define comp
        (plambda ((t type))
          (lambda ((f (subr pure (t) t))
                   (g (subr pure (t) t)))
                  (lambda ((x t)) (f (g x))))))
comp = <subr> : (poly ((t type))
                      (subr pure ((subr pure (t) t)
                                  (subr pure (t) t))
                                 (subr pure (t) t)))
              ! pure
```

This feature of *FX* is called *polymorphism*. (The name **plambda** comes from "Polymorphic Lambda.") Polymorphism permits an expression to be abstracted over types, effects, and regions. The subroutine **comp** can be used in precisely the same way as **compose** – the type parameter is supplied automatically by *FX* using a mechanism called *implicit projection*. The following example shows how **comp** can be useful on different types of values:

```
FX => ((comp fib (lambda ((x int)) (+ x 1))) 4)
5 : int ! pure
FX => ((comp not? not?) #t)
#t : bool ! pure
```

## Lists are Defined with Recursive Types

The range of expressible types is quite large in *FX* since it is possible to define *recursive* types with the **dletrec** special form. For example, in *FX* the **listof** type is defined in terms of a recursive **pairof** type. A subroutine that uses both polymorphism and recusive types is the **mapcar** subroutine:

```
FX => (define mapcar
        (plambda ((t1 type) (t2 type) (r region) (e effect))
          (lambda ((f (subr e (t1) t2))
                   (input (listof t1 r)))
            (the (maxeff (alloc r) (read r) e)
                 (listof t2 r)
                 (if (null? input)
                     ()
                     ((proj cons r)
                      (f (car input))
                      (mapcar f (cdr input))))))))
mapcar = <subr>
      : (poly ((t1 type) (t2 type) (r region) (e effect))
              (dletrec ((#1 (pairof t1 #1 r))
                        (#2 (pairof t2 #2 r)))
                (subr (maxeff (alloc r) (read r) e)
                      ((subr e (t1) t2) #1)
```

```
                                #2)))
         ! pure
FX => (mapcar (lambda ((x int)) (+ x 1)) (list 1 2 3))
(2 3 4) : (dletrec ((#1 (pairof int #1 @=))) #1) ! pure
```

## Types can be Abbreviated

Because complicated types may be cumbersome to write *FX* provides a type synonym facility. You can introduce type synonyms in an expression with the **plet** construct, as in:

```
FX => (define comp
         (plambda ((t type))
           (plet ((func (subr pure (t) t)))
             (lambda ((f func) (g func))
                        (lambda ((x t)) (f (g x)))))))
comp = <subr> : (poly ((t type))
                   (subr pure ((subr pure (t) t)
                               (subr pure (t) t))
                              (subr pure (t) t)))
              ! pure
```

You can also introduce type synonyms at top-level, with the **pdefine** top-level special form:

```
FX => (pdefine int-subr (subr pure (int int) int))
int-subr = (subr pure (int int) int) :: type
```

After a **pdefine** form is evaluated, the *FX* interpreter prints out the name of the variable defined, the description to which it is bound, and after the " :: ", the *kind* of the variable. Kinds are the "types" of descriptions.

You can also define recursive types at top-level; for instance, you could define the abstract syntax of a simple expression language as:

```
FX => (pdefine expr (oneof ((constant int)
                            (identifier symbol)
                            (add (pairof expr expr @=)))
                           @=))
expr = (dletrec ((#1 (oneof ((constant int)
                             (identifier symbol)
```

```
                                    (add (pairof #1 #1 @=)))
                                   @=)))
                 #1) :: type
```

Here, oneof is a standard *FX* type constructor. The idea is that a value of type expr can be either an integer constant (in which case it will be "tagged" by constant), an identifier represented by a symbol (yet another standard *FX* type) or an addition of two other expressions.

## A Simple Evaluator

In order to define an evaluator (*i.e.*, a function which maps expr values to integers) for our new tiny language, we need one more type definition, namely for the store in which the values of identifiers are kept. Here it is:

```
FX => (pdefine store (subr pure (symbol) int))
store = (subr pure (symbol) int) :: type
```

A function of this type will map each identifier (recall that they are in fact implemented by symbols) to its integer value.

The definition of our evaluation function eval is now easy. It take two arguments: an expression e and a store s in which every identifier used in e has a value. We will suppose that there are no unbound identıfiers.

```
FX => (define (eval (e expr) (s store))
        (the pure int
             (tagcase e
               (constant e)
               (identifier (s e))
               (add (+ (eval (car e) s)
                       (eval (cdr e) s))))))
eval = <subr>
     : (dletrec ((#1 (oneof ((constant int)
                             (identifier symbol)
                             (add (pairof #1 #1 @=)))
                            @=)))
                (subr pure
                      (#1 (subr pure (symbol) int))
                      int))
     ! pure
```

The **eval** function is a pure subroutine which maps an expr, which is represented by the complicated recursive type **#1**, and a **store** to a value of type **int**.

The **tagcase** special form is used to dispatch on a oneof value according to its tag. Inside each clause of such a form, the variable **e** denotes the contents of the oneof value; for instance, inside the **identifier** clause, **e** represents the symbol corresponding to the **identifier** value. This is why we apply the store **s** to **e** to get its integer value. It is also interesting to note that the recursive calls to **eval** used to compute the addends of an add expression have no side-effects (the **eval** function is **pure**). Therefore, the *FX* compiler may schedule them to run in parallel safely.

Let us check that our definition works.

```
FX => (define x-plus-1 (one expr add
                            (cons (one expr identifier 'x)
                                  (one expr constant 1))))
x-plus-1 = (add (identifier . x) constant . 1)
         : (dletrec ((#1 (oneof ((constant int)
                                  (identifier symbol)
                                  (add (pairof #1 #1 @=)))
                                 @=)))
                     #1)
         ! pure
FX => (eval x-plus-1 (lambda ((s symbol))
                        (if (eq? s 'x) 3 0)))
4 : int ! pure
```

We first define the expression x-plus-1 corresponding to the addition of the identifier 'x (symbol literals are distinguished by the use of a quote character) and the constant 1; one is the special form defined in *FX* to construct an value whose type is a oneof. The real test is then to evaluate x-plus-1 in a store which binds 'x to the value 3; the result is, of course, 4.

## Effects can be Masked

Effects that are not observable outside of an expression can be *masked* by the effect system. For example, an assignment to a formal subroutine parameter may not be reported as part of the latent effect of the subroutine if it cannot be observed by the caller:

```
FX => (define (f (x int @local))
              (set! x (+ x 1))
              (* x x))
F = <subr> :  (subr pure (int) int)  !  pure
FX => (f 10)
121  :  int  !  pure
```

Similarly, a subroutine that constructs a circular list has only an `alloc` effect, even though it mutates the list after allocating it, because this mutation cannot be observed by the caller:

```
FX => (define circular-list
        (plambda ((r region))
          (plambda ((t type))
            (lambda ((init t))
                   (let ((l ((proj list r) init)))
                        (set-cdr! l l)
                        l)))))
circular-list = <subr>
              :  (poly ((r region))
                   (poly ((t type))
                        (dletrec ((#1 (pairof t #1 r)))
                                 (subr (alloc r) (t) #1))))
              !  pure
fx> (lambda () ((proj circular-list @green) 5))
<subr>  :  (dletrec ((#1 (pairof int #1 @green)))
                   (subr (alloc @green) () #1))
        !  pure
fx> (lambda () (circular-list 5))
<subr>  :  (dletrec ((#1 (pairof int #1 @=)))
                   (subr pure () #1))
        !  pure
```

As this example shows, the effect of creating a `@green` circular list is `(alloc @green)`, and the effect of creating an immutable circular list is pure. Effect masking is more thoroughly described in Section 2.3.

This ends our short primer on *FX*. We hope that you have a sense of how the *FX* language can be used. The best way to learn more about *FX* is to try writing a few programs. The rest of this manual contains all of the information that you need to write programs on your own.

Have fun, and good effects!

Chapter 2

# The FX Kernel

The *FX* Kernel is a simple programming language that is the basis of the *FX* programming language. All of the constructs in the *FX* language can be directly explained by rewriting them into the simpler *FX* Kernel language. Thus, the *FX* Kernel forms the core of the *FX* language from the point of view of both the *FX* application programmer and the *FX* language implementor.

The *FX* Kernel has three language levels each with its own set of expressions: *value expressions, description expressions* and *kind expressions.* In the simplest terms, programs are value (or ordinary) expressions, types are descriptions, and kinds are the "types of types".

- Value expressions form the lowest level of the language. Programs and literals (*e.g.* #t) are examples of value expressions.
- Descriptions form the second level of the language. There are four kinds of descriptions: *region*, *effect*, *type* descriptions and description functions. As the name suggests, descriptions describe value expressions – in particular, every legal value expression has both a type and an effect description. Region descriptions are used as components of effect descriptions.
- Kinds form the third and highest level of the language. Kinds are the "types" of descriptions, and every legal description expression has a kind.

*FX* is a block-structured, lexically-scoped language, like Scheme or CommonLISP. Whenever a variable is used, it refers to the inner-most lexical binding of that variable. A variable may stand for a description or a value, but may not be bound simultaneously to both a description and a value.

## 2.1 Kind Expressions

### 2.1.1 Meta-notation for Kinds

Kind expressions have the meta-notation *Kexp*. A kind expression is either a kind constant or a kind constructor expression.

To express the idea that a description expression has some kind, we use a double colon.

$$Dexp \;::\; Kexp$$

should be read: "The description expression *Dexp has kind Kexp*." We will at times avail ourselves of the shorthand notation

$$Dexp_1,\ldots,Dexp_n \;::\; Kexp$$

to mean that each of the $Dexp_i$ $(1 \leq i \leq n)$ is of kind *Kexp*.

### 2.1.2 Kind Constants

*FX* has three kind constants:

- `region` is the kind of a description which describes an area of memory (*e.g.* `@=` :: `region`).
- `effect` is the kind of a description which describes the side-effects of a computation (*e.g.* pure :: `effect`).
- type is the kind of a description which describes a set of values (*e.g.* bool :: type).

### 2.1.3 Kind Constructors

dfunc expressions provide a way to build new kinds; dfunc is a kind *constructor*.

$$(\texttt{dfunc}\ (Kexp_1 \ldots Kexp_n)\ Kexp)$$

Kinds built with `dfunc` are the kinds of *description functions* which map descriptions to descriptions. A description function which returns a type is called a type *constructor* because it provides a way to build a new type (*e.g.*, `ref :: (dfunc (type region) type)`). Effect and region constructors arise in the same way.

One creates description functions with the `dlambda` description special form described on page 23. One uses description functions by applying them as described on page 24.

## 2.2 Description Expressions

Description expressions are used to describe *FX* values and program expressions. Every legal description expression has a kind (e.g. the description expression `(ref bool @=)` has kind `type`.)

### 2.2.1 Meta-notation for Descriptions

Description variables have the meta-notation *d*, and description expressions have the meta-notation *Dexp*.

### 2.2.2 Variables

The programmer may use any unreserved identifier as a description variable. (See page xiii for a list of reserved identifiers.)

A description variable denotes the description it is bound to in the surrounding bindings.

### 2.2.3 Regions

A *region* represents a set of locations in the store. The programmer may think of a region as an area of memory (though a region may not actually be a contiguous set of memory locations). The compiler may use regions to determine whether two program expressions *interfere*, *i.e.* whether they may cause and/or observe changes to common data. Since one cannot determine interference patterns exactly for every data value without running the program, the *FX* compiler makes the conservative assumption that expressions with side effects (see below) in the same region do interfere.

## Meta-notation for Regions

The meta-notation for description variables of kind **region** is *r*. Region expressions have the meta-notation *Rexp*. A region expression is a region variable, a region constant, or a region constructor expression.

## Region Constants

All region constants begin with the special character **@** (read "at"). There is one special region called the *immutable region* whose name is **@=** (read "at-equal" or "the immutable region"). This region has the property that values in it can never be changed. There is an infinite supply of other region constants which are mutable; their names are of the form **@***identifier*. All region constants denote disjoint sets of locations.

## Region Constructors

The only operation on regions is **runion**.

(runion $Rexp_1$ $Rexp_2 \ldots$)

**Semantics:** The runion operation forms the set union over all the regions denoted by the $Rexp_i$; the result is a region. The runion of just one region is equivalent to that region.

runion expressions are *flattened*, *i.e.* inner runion expressions are replaced with the regions of the set to which they correspond. Duplicates are ignored and order is not significant. For example, `(runion @a (runion @a @b))` is equivalent to `(runion @b @a)`.

**Kind Information:** The kind of an runion expresssion is region.

**Example:**

```
;; Composing "colored" regions.
;;
(runion @blue @red @yellow)
```

### 2.2.4 Effects

The *effect* of an expression is a static description of the observable store operations which may be performed during the evaluation of the expression. Store operations performed by an expression are observable when they are performed on store locations that are accessible outside of the scope of the expression.

*FX* uses effects to discover expression scheduling constraints. *FX* will constrain two expressions to be executed serially only if the expressions interfere with one another. Two program expressions *interfere* if one expression writes a region of the store that the other expression reads or writes.

*FX* also uses the property that expressions with the pure effect are *referentially transparent*. Informally speaking, an expression is referentially transparent if two occurances of the expression in the same scope can be replaced by a single instance of the expression. Thus, when an expression is referentially transparent both static common subexpression elimination and dynamic memoization can be applied to it.

#### Meta-notation for Effects

Effect variables have the meta-notation *e*, and effect expressions have the meta-notation *Eexp*. An effect expression is an effect variable, a simple effect, or an effect constructor expression.

#### Simple Effects

*FX* describes effects in terms of three sorts of operations on the store:

- One may *allocate* and initialize a location in the store, *e.g.* by making a binding for a variable in some region.
- One may *read* a location in the store, *e.g.* by referencing a variable in some region.
- One may *write* a location in the store, *e.g.* by assigning to a variable in some region.

The corresponding effects on a region *Rexp* are written:

(`alloc` *Rexp*)
(`read` *Rexp*)
(`write` *Rexp*)

An expression which has no effect or dependence on the store is said to have effect `pure`.

Recall that the special region `@=` is immutable. Therefore, it is a static effect error for an expresion to have effect (`write @=`). Since no value in the immutable region may ever be changed, the operations of allocating in and reading from `@=` cannot be observed. Thus, (`alloc @=`) and (`read @=`) are both equivalent to `pure`.

### Effect Constructors

The only operation on effects is `maxeff`.

(maxeff *Eexp*$_1$ ...)

**Semantics:** The maxeff operation can be thought of as constructing the *set* or *combination* of all effects corresponding to the effect expressions supplied as arguments; the result is an effect. The maxeff of just one effect is equivalent to that effect.

maxeff expressions are flattened in the same way as runion expressions; inner maxeff expressions are replaced with the effects from the set to which they correspond. Duplicates are ignored and order is not significant. The simpler expression pure is an abbreviation for the empty set of effects (maxeff).

The simple effects have a distributive property over runion. For instance, (alloc (runion @a @b)) can be rewritten (maxeff (alloc @a) (alloc @b)). We will always use this latter version.

**Kind Information:** The kind of a maxeff expression is effect.

**Example:**

```
;; The most complex effect on @foo.
;;
(maxeff (alloc @foo) (read @foo) (write @foo))
```

### 2.2.5 Types

In *FX*, a *type* denotes a collection of values. A value has some particular type if it is in the collection of values denoted by the type.

#### Meta-notation for Types

Type variables have the meta-notation *t*. Type expressions have the meta-notation *Texp*. A type expression is a type variable, a type constant, or a type constructor expression.

#### Type Constants

The builtin types are:

`bool` the type containing the two boolean values `#t` and `#f` for *true* and *false*. The type of the predicate of a conditional, *i.e.* `if`, is `bool`.

`unit` the type containing the single value `#u`. The `unit` type is useful as the return type of subroutines which are called solely for their side-effects and which do not compute a useful return value.

#### Type Constructors

There are three builtin ways to build new types from old ones. They are described on the following pages: `subr`, `poly` and `ref`.

(**subr** *Eexp* (*Texp*$_1$...) *Texp*)

**Semantics:** This is the type of a subroutine created by the `lambda` expression.

*Eexp* is the *latent effect* of the subroutine; that is, upon application, evaluation of the subroutine will have an effect of *Eexp*. The *Texp*$_i$ are the types of the parameters. *Texp* is the type of the value which the subroutine will return. (See the descriptions of `lambda` on page 34 and ordinary application on page 36.)

**Kind Information:** The kind of a **subr** expression is **type**.

**Example:**

```
;; The type of the identity function on booleans.
;;
(subr pure (bool) bool)
```

(poly (($d_1$ *Kexp*$_1$)...) *Texp*)

The $d_i$ must all be distinct.

**Semantics:** This is the type of a polymorphic value created by the plambda expression.

The description variables $d_i$ are bound variables; their kinds are given by the *Kexp*$_i$. When a polymorphic value is projected (either explicitly or implicitly), the description arguments are bound to the $d_i$ and the result of the projection is a value whose type is *Texp* with the $d_i$ substituted by the corresponding description arguments. (See the descriptions of the plambda expression on page 39, projection on page 40 and implicit projection on page 41.)

**Kind Information:** The kind of a poly expression is type.

**Example:**

```
;; The type of the polymorphic identity function.
;;
(poly ((t type))
      (subr pure (t) t))
```

The type of a value which is a *reference* to a location in region *Rexp* containing a value of type *Texp* is:

(`ref` *Texp Rexp*) :: **type**

The kind of `ref` is:

`ref` :: (**dfunc** (**type region**) **type**)

### 2.2.6 General Description Expressions

There are three description expressions which provide general ways of manipulating descriptions (of any kind). These expressions are used to define and apply description functions, and to build recursive descriptions. These expressions are described on the following pages: `dlambda`, Description Application and `dletrec`.

(**dlambda** (($d_1$ $Kexp_1$)...($d_n$ $Kexp_n$)) $Dexp_{body}$)

The $d_i$ must all be distinct.

**Semantics:** Just as **lambda** provides a means of abstracting program code over ordinary variables to make subroutines, the **dlambda** description expression provides a means of abstracting a description expression over description variables to make description functions. (See also the definition of description application on page 24.)

**Kind Information:** The kind of a description function created with **dlambda**, assuming that the kind of $Dexp_{body}$ is $Kexp_{body}$, is:

(**dfunc** ($Kexp_1$...) $Kexp_{body}$)

**Examples:**

```
;; An effect constructor which is abstracted over a region.
;;
(dlambda ((r region))
         (maxeff (alloc r) (read r) (write r)))

;; A constructor of types of subroutines which have
;; every possible side effect on some region.  The caller
;; should specify the argument type (there is only one
;; argument) and the return type of the subroutine.
;;
(dlambda ((arg-type type) (ret-type type) (r region))
         (subr (maxeff (alloc r) (read r) (write r))
               (arg-type)
               ret-type))
```

$(\boldsymbol{Dexp}_{func}\ \boldsymbol{Dexp}_1 \ldots)$

**Semantics:** $\boldsymbol{Dexp}_{func}$ must evaluate to a description function (see the description of **dlambda** on page 23). In particular, the kind of $\boldsymbol{Dexp}_{func}$ must be (dfunc ($\boldsymbol{Kexp}_1 \ldots$) $\boldsymbol{Kexp}_{body}$). The $\boldsymbol{Dexp}_i$ are the actual parameters, or arguments, to the function. Each argument expression must be of the proper kind; *i.e.* $\boldsymbol{Dexp}_i$ must be of kind $\boldsymbol{Kexp}_i$. The number of actual parameters must be the same as the number of formal parameters. When the description function is applied to its arguments, the description expression which is its body is returned with the arguments substituted for the formal parameters.

**Kind Information:** The kind of a description application is $\boldsymbol{Kexp}_{body}$, the kind of the body of the **dlambda** form which defines the description function.

**Examples:**

```
;; A synonym for the type of the identity function on booleans.
;;
((dlambda ((t type))
          (subr pure (t) t))
 bool)

;; A complicated effect.
;;
((dlambda ((r region))
          (maxeff (alloc r) (read r) (write r)))
 (runion @green @blue @red @purple))
```

**(dletrec** $((d_1\ \mathit{Dexp}_1)\ldots)\ \mathit{Dexp}_{body}$**)**

The $d_i$ must all be distinct.

**Semantics:** First, all of the $d_i$ are made available. Then, the $\mathit{Dexp}_i$ are successively evaluated and bound to the corresponding $d_i$. This order is important because it allows any of the $\mathit{Dexp}_j$ to refer to some $d_i$, thus providing for mutual recursion. This process is subject to the following restriction: all description variables that can be recursively reached, in the graph of description variable usage, from an expression $\mathit{Dexp}_i$ must be defined *before* the processing of the binding for $d_i$, unless their kinds are type. Typically, the $\mathit{Dexp}_j$ are recursive type descriptions and so there is no problem. It is a static type error if any of the $d_i$ *is defined as itself, either directly or* indirectly.

The value of the **dletrec** expression is the body, $\mathit{Dexp}_{body}$, evaluated with the (possibly recursive) definitions of the $d_i$ substituted for the $d_i$.

**Kind Information:** The kind of a **dletrec** description special form is the kind of $\mathit{Dexp}_{body}$.

**Example:**

```
;; Example of list.
;;
(dletrec ((list-bool (pairof bool list-bool @bool)))
         list-bool)
```

### 2.2.7 Description Inclusion

Some description expressions are more constrained versions of others. A more constrained description is said to be *included in* (a *subdescription of*) the less constrained one. For instance, a region is a set of memory locations. If every location in one region is also in another, then the first is a subregion of the second, and is said to be *included in* the second. We can also define subeffects and subtypes. Two description expressions are *interconvertible* if and only if each is included in the other; two description expressions that are interconvertible denote the same description.

One description expression can only be a subdescription of another if both are of the same kind: a type cannot be included in a region since types and regions describe different things.

The *least upper bound* of a set of descriptions of the same kind is the least description of that kind that includes all the members of the set. The `runion` operation computes the least upper bound of a set of regions, and `maxeff` computes the least upper bound of a set of effects. No least upper bound operation is provided for types because certain sets of types do not have a least upper bound.

The *maximum* description of a set of descriptions (all of the same kind) is the element of the set that includes all the members of the set. Since not every set of descriptions has a maximum, *FX* provides no way to express the maximum of a set of descriptions.

#### Region Inclusion

Recall that `runion` expressions are canonicalized (by flattening) and that the `runion` of one region is equivalent to this region.

The region denoted by $Rexp_1$ is a subregion of the region denoted by $Rexp_2$ iff (if and only if) the set of regions in $Rexp_1$ is a subset of the set of regions in $Rexp_2$.

#### Effect Inclusion

Recall that `maxeff` expressions are canonicalized (by flattening) and that the `maxeff` of one effect is equivalent to this effect.

These rules depend upon the rules for region inclusion:

- pure is a subeffect of any other effect and is a shorthand for (`maxeff`).

- (alloc $Rexp_1$) is included in (alloc $Rexp_2$) iff $Rexp_1$ is included in $Rexp_2$. The analogous rules hold for read and write.
- (maxeff $Eexp_{11} \ldots Eexp_{1n}$) is included in (maxeff $Eexp_{21} \ldots Eexp_{2m}$) iff every $Eexp_{1i}$ is contained in *some* $Eexp_{2j}$.

## Type Inclusion

There is no inclusion between the base types bool and unit. So we need only describe the way inclusion works with respect to the type constructors:

- Suppose

$$\begin{aligned} t_1 &\equiv (\texttt{subr}\ Eexp_1(Texp_{11} \ldots Texp_{1n})\ Texp_{1rtn}) \\ t_2 &\equiv (\texttt{subr}\ Eexp_2(Texp_{21} \ldots Texp_{2m})\ Texp_{2rtn}) \end{aligned}$$

  $t_1$ is a *subtype* of $t_2$ iff

  1. $m = n$,
  2. $Eexp_1$ is a subeffect of $Eexp_2$,
  3. $Texp_{2i}$ is a subtype of $Texp_{1i}$ (for $1 \leq i \leq n$), and
  4. $Texp_{1rtn}$ is a subtype of $Texp_{2rtn}$.

  Notice that if $t_1$ is a subtype of $t_2$, then the types of the arguments of $t_1$ are *supertypes* of the types of the arguments of $t_2$.

  Rationale: Imagine a program being passed a subroutine as an argument. If you pass such a program a subroutine whose type is a subtype of the expected one, the program should still be able to work properly since the program could handle the larger type. The effect cannot be larger than originally expected; the program should still be able to pass the subroutine *at least* the same argument types as expected (one may extend the set of arguments accepted but may not restrict it); and the subroutine should return a subtype of the return type expected so that the calling program can handle the result.

- Suppose

$$\begin{aligned} t_1 &\equiv (\texttt{poly}\ ((d_{11} Kexp_{11}) \ldots (d_{1n} Kexp_{1n}))\ Texp_1) \\ t_2 &\equiv (\texttt{poly}\ ((d_{21} Kexp_{21}) \ldots (d_{2m} Kexp_{2m}))\ Texp_2) \end{aligned}$$

$t_1$ can only be a subtype of $t_2$ if $n = m$. The particular names chosen for formal parameters do not matter, so we can pick new unused names $d_1' \ldots d_n'$ and substitute $d_i'$ for $d_{1i}$ in $Texp_1$ and for $d_{2i}$ in $Texp_2$. This process of renaming bound variables with unused names is called *alpha-conversion*.

Then, $t_1$ is a subtype of $t_2$ iff

1. $Kexp_{1i} = Kexp_{2i}$ (for $1 \leq i \leq n$) and
2. $Texp_1$ is a subtype of $Texp_2$.

- Suppose

$$\begin{aligned} t_1 &\equiv (\texttt{ref}\ Texp_1 Rexp_1) \\ t_2 &\equiv (\texttt{ref}\ Texp_2 Rexp_2) \end{aligned}$$

$t_1$ is a subtype of $t_2$ iff either

1. $Rexp_1$ is a subregion of $Rexp_2$ and $Texp_1$ is interconvertible with $Texp_2$, or
2. $Rexp_1 = Rexp_2 = \texttt{@=}$ and $Texp_1$ is a subtype of $Texp_2$.

Rationale: Consider a subroutine which expects an argument of some reference type. If the subroutine expects the reference to be in a mutable region, then it is perfectly reasonable for the subroutine to write to the location denoted by the reference. Now if we pass this subroutine a reference to a value of some subtype of the expected type, the subroutine could mutate the reference so that it refers to a value of a larger type (namely the type specified inside the expected reference type). But this would be a type error! Therefore, we require the type parts of mutable references to be interconvertible in the subtyping rules.

The assignment problem cannot arise if the reference is located in the immutable region `@=`; the natural subtyping rule applies here.

## General Description Inclusion

Notice that the following inclusion rules are symmetric. That is if one description function or description application is included in another, then the two are, in fact, interconvertible.

- Suppose

$$f \equiv (\texttt{dlambda}\ ((d_1\ Kexp_1) \ldots (d_n\ Kexp_n))\ (Dexp\ d_1 \ldots d_n))$$

  Then $f$ is interconvertible with $Dexp$ iff none of the $d_i$ occurs unbound in $Dexp$. This rule is called *eta-conversion*.

- Suppose we have the two description functions

$$\begin{aligned} f_1 &\equiv (\texttt{dlambda}\ ((d_{11}\ Kexp_{11}) \ldots (d_{1n}\ Kexp_{1n}))\ Dexp_{body1}) \\ f_2 &\equiv (\texttt{dlambda}\ ((d_{21}\ Kexp_{21}) \ldots (d_{2m}\ Kexp_{2m}))\ Dexp_{body2}) \end{aligned}$$

  $f_1$ is included in $f_2$ iff

  1. $n = m$,
  2. $Kexp_{1i} = Kexp_{2i}$ (for $1 \leq i \leq n$), and
  3. $Dexp_{body1}$ is interconvertible with $Dexp_{body2}$ after alpha-conversion. (Alpha-conversion is defined on page 28.)

- Suppose we have the two description applications

$$\begin{aligned} d_1 &\equiv (Dexp_{11}\ Dexp_{12} \ldots Dexp_{1n}) \\ d_2 &\equiv (Dexp_{21}\ Dexp_{22} \ldots Dexp_{2m}) \end{aligned}$$

  $d_1$ is included in $d_2$ iff

  1. $n = m$ and
  2. $Dexp_{1i}$ is interconvertible to $Dexp_{2i}$ (for $1 \leq i \leq n$).

## 2.3 Value Expressions

Value expressions are the bottom of our language hierarchy; this is where the actual computation gets done.

### 2.3..1 Meta-notation for Value Expressions

Ordinary variables (variables denoting actual computational values rather than descriptions) have the meta-notation *var*. Ordinary expressions, or *value expressions*, have the meta-notation *exp*. A value expression is an ordinary variable, a builtin literal or a compound expression.

We often want to express the fact that a value expression or a set of value expressions has some type. (Just as we wanted to express that a description expression had a kind. See page 12.)

$$exp : Texp$$
$$exp_1,\ldots,exp_n : Texp$$

This notation means that *exp has type Texp* or, in the second case, that each $exp_i$ is of type *Texp*.

We also want to express the fact that a value expression or a set of value expressions has some effect.

$$exp \ ! \ Eexp$$
$$exp_1,\ldots,exp_n \ ! \ Eexp$$

This notation means that *exp has effect Eexp* or, in the second case, that each $exp_i$ has effect *Eexp*.

### 2.3..2 Effect Masking

The effect of a value expression derives from operations that the expression performs on the store: *i.e.* allocating, reading, and writing. But even if a value expression performs certain operations on the store, the compiler may be able to prove that those operations cannot interfere with other expressions. If this is the case, then the effect system will *mask* effects which derive from those operations. For example, an assignment to a formal subroutine parameter need not be reported as part of the latent effect of the subroutine since it alters a part of the store known only to an invocation of the subroutine and thus cannot interfere with expressions outside of the subroutine. The rule for effect masking is:

> If a value expression has effects on some region *r*, and if *r* does not appear in the type of any free ordinary variables in the expression, then any `read` or `write` effect on *r* is masked; furthermore, if *r* does not appear in the type of the whole expression, then any `alloc` effect on *r* is also masked.

A variable is *free* in a value expression if it appears in but is not bound in the expression. See Appendix B for a formal definition of free variables.

### 2.3.1 Builtin Literals

The builtin constants are `#t` for the boolean value *true*, `#f` for the boolean value *false*, and `#u` for the single value of type `unit` (signifying "nothing" or "done").

As expressions, they evaluate to themselves and have a pure effect.

### 2.3.2 Variables

The programmer may use any unreserved identifier as a variable. (See page xiii for a list of reserved identifiers.) When a variable is introduced in a `letrec` or `lambda` expression, the programmer may specify that the value corresponding to the variable will be in some region. If no region is specified, the value is assumed to be in the region `@=`. If a variable is in region `@=`, then the value of that variable may not be changed, *i.e.* may not be the first argument to a `set!` expression.

A variable evaluates to the value it denotes and has a `read` effect on the region where it is located, which reduces to `pure` if the region is `@=`.

### 2.3.3 Compound Expressions

The following pages document the compound expressions (or special forms) defined by the *FX* kernel. Each compound expression performs some computation and returns a value. Hence, these expressions are called *value expressions*: `begin`, `the`, `lambda`, Application, `letrec`, `plambda`, `proj`, Implicit Projection, `plet`, `pletrec`, `if` and `set!`.

(**begin** $exp_1$ $exp_2 \ldots exp_n$)

**Semantics:** The expressions in a **begin** expression are evaluated in order, left to right. The value of the **begin** expression is the value of the last expression, $exp_n$.

**Type Information:** The type of a **begin** expression is the type of the last expression, $exp_n$.

**Effect Information:** The effect of a **begin** expression is computed by performing effect masking on the `maxeff` of the effects of the expressions in the sequence, the $exp_i$.

**Example:**

```
;; Call foo and then bar (which probably have
;; side-effects).  Return the value returned by bar.
;;
(begin (foo)
       (bar))
```

(**the** [*Eexp*] *Texp exp*)

**the** expressions provides a way to declare the type or, possibly, the effect of some value expression.

**the** expressions are useful as a form of documentation or as a means of coercing *exp* to a higher effect and/or to a supertype of its actual type. One might make use of this feature to prevent code from depending on the current return value of a stub subroutine, *i.e.* **the** can be used to assert that the subroutine has a particular return type and effect which is more complicated than the real type and effect. (**the** is *not* a type loophole.)

**Semantics:** The value of a **the** expression is the value *exp*. *exp* must have a type which is a subtype of *Texp* and an effect less than or equal to *Eexp* (if *Eexp* is given).

**Type Information:** The type of a **the** expression is *Texp*.

**Effect Information:** The effect of a **the** expression is *Eexp* if it is supplied. Otherwise, it is the effect of *exp*.

**Example:**

```
;; Simulates a write effect on @foo.
;;
(the (write @foo) int 0)
```

(lambda (($var_1$ $Texp_1$ [$Rexp_1$])...($var_n$ $Texp_n$ [$Rexp_n$]))
$exp_1$ $exp_2$... $exp_m$)

The body of the lambda expression, $exp_1$ $exp_2$... $exp_m$, is treated as though (begin $exp_1$ $exp_2$... $exp_m$) is written.

The $var_i$ must all be distinct.

**Semantics:** lambda provides a way of *abstracting* program expressions over ordinary variables to make a subroutine value or *closure* which is the value of the lambda expression. The subroutine takes $n$ arguments and, when applied to $n$ arguments each of the proper type, *i.e.* the type of argument $i$ is a subtype of $Texp_i$, returns the value of its body evaluated with the formal parameters bound to the argument values. If no region is specified for a formal parameter, then @= is assumed, and the body of the subroutine must not contain any assignments to that formal. If a region *is* specified for a formal, then a new location in that region is allocated and given the argument as its value; the formal is then bound to this location. Assignments to such formals are allowed, provided the region is not @=.

*FX* uses "call-by-sharing" semantics : a set! on a formal parameter only changes the binding of the formal and does not change variable bindings in the caller's environment.

**Type Information:** If the type of $exp_m$ is $Texp_{body}$, then the type of the subroutine value created is:

(subr $Eexp$ ($Texp_1$ $Texp_2$... $Texp_n$) $Texp_{body}$).

where $Eexp$ is the *latent effect* of the subroutine. $Eexp$ is computed by performing effect masking on the maxeff of (alloc $Rexp_i$) and the effects of the $exp_j$.

**Effect Information:** The effect of a lambda expression is pure, *i.e.* creating a subroutine value is a pure operation. The effect $Eexp$ will be used in determining the effect of an application involving the subroutine value.

**Examples:**

```
;; A nullary subroutine which returns the value of x.
;;
(lambda () x)

;; The "apply-twice" functional on booleans.
;;
(lambda ((g (subr pure (bool) bool))
         (x bool))
        (g (g x)))
```

$(exp\ exp_1 \ldots exp_n)$

**Semantics:** The expressions *exp* and $exp_i$ are evaluated sequentially, from left to right. If *exp* is polymorphic, then implicit projection is used to obtain a subroutine value (see page 41). The $exp_i$ are the actual parameters, or arguments, to the subroutine. The formal parameters of the subroutine are allocated and bound to the argument values, and the body of the subroutine is then evaluated in the resulting environment. The value of the application expression is the result of the evaluation of the subroutine body.

**Type Information:** *exp* must have type (subr *Eexp* ($Texp_1 \ldots Texp_n$) $Texp_{ret}$). The number of actual and formal parameters must be the same. The type of each $exp_i$ must be a subtype of $Texp_i$.

The type of the application expression is the return type, $Texp_{ret}$, of the subroutine.

**Effect Information:** The effect of the application expression is computed by performing effect masking on the `maxeff` *of the latent effect of the* subroutine, *Eexp*, and the effects of *exp* and the $exp_i$.

**Examples:**

```
;; A synonym for #f.
;;
(not? #t)

;; ...  and another for #t.
;;
(and? #t (not? #f))
```

(**letrec** ((*var*$_1$ *exp*$_1$ [*Rexp*$_1$])...) *exp*$_{1b}$ *exp*$_{2b}$...)

The body of the **letrec** expression, *exp*$_{1b}$ *exp*$_{2b}$..., is treated as though (**begin** *exp*$_{1b}$ *exp*$_{2b}$...) is written.

The *var*$_i$ must all be distinct.

The expression to which *var*$_i$ is bound must be either

- a (possibly polymorphic) syntactic subroutine, *i.e.*, a (possibly empty) nest of **plambda** forms followed by a subroutine form. If the subroutine is recursive, its body must be of the form (**the** *Eexp*$_i$ *Texp*$_i$ *exp*$_i$).
- a non-recursive, non-subroutine value expression.

A **the** expression is necessary since it is not possible, in general, to determine the types and effects of arbitrary recursive expressions. The programmer must supply them.

**Semantics:** First, all of the *var*$_i$ are allocated, each in *Rexp*$_i$ (or **@=** if unspecified). Then, the *exp*$_i$ are successively evaluated and bound to the corresponding *var*$_i$. This order is important because it allows any of the *exp*$_j$ to refer to some *var*$_i$, thus providing for mutual recursion. This process is subject the following restriction: all variables that can be recursively reached, in the graph of variable usage, from an expression *exp*$_i$ which is not a (possibly polymorphic) syntactic subroutine, must be defined *before* the processing of the binding for *var*$_i$. Since evaluation of a (possibly polymorphic) syntactic subroutine does not imply the evaluation of the body of the lambda expression, this restriction does not apply to the bindings involving such subroutines. Typically, the *exp*$_j$ are **lambda** expressions and therefore no problem exists.

After all the bindings are done, the body of the **letrec** expression is evaluated in the environment with these additional bindings.

A subroutine is *tail-recursive* if all of the values returned from recursive calls are themselves returned without further computation. *FX* guarantees that a properly tail-recursive subroutine will be translated to an iteration. Since do loops can be implemented with recursive subroutine calls which the compiler recognizes as iterative, the *FX* kernel need not have any separate looping expressions.

**Type Information:** The type of *var*$_i$ is the type of *exp*$_i$. The type of the **letrec** expression is the type of its body.

**Effect Information:** The effect of a `letrec` expression is computed by performing effect masking on the `maxeff` of (`alloc` *Rexp*$_i$) and the effects of the *exp*$_i$ and the *exp*$_{jb}$.

**Example:**

```
;; The traditional factorial program.
;;
(letrec ((fact (lambda ((x int))
                       (the pure int
                            (if (= x 0) 1
                                (* x (fact (- x 1))))))))
        (fact 10))
```

(**plambda** (($d_1$ *Kexp*$_1$)...) *exp*)

The $d_i$ must all be distinct.

**Semantics:** Just as **lambda** provides a means of abstracting expressions over ordinary variables, **plambda** provides a means of abstracting expressions over *description* variables. The kind of these variables must be **type**, **region**, **effect** or any **dfunc** expression whose eventual kind is **type**.

The evaluation of **plambda** expression yields a *polymorphic* value that takes *n* arguments and, when *projected* onto *n* descriptions each of the proper kind, *i.e.* argument *i* is of kind *Kexp*$_i$, returns the value of *exp* evaluated with the formal parameters bound to the argument descriptions. (See the description of the **proj** expression on page 40 and the description of implicit projection on page 41.)

The body of a **plambda** expression, *exp*, must be a pure expression. Because of this restriction, the body of a **plambda** expression can be evaluated when the **plambda** expression is evaluated rather than each time it is projected. Every *FX* implementation guarantees that projection has no run-time cost.

**Type Information:** Assuming that the type of *exp* is *Texp*, the type of a polymorphic value defined as above is (poly (($d_1$ *Kexp*$_1$)...) *Texp*).

**Effect Information:** The effect of a **plambda** expression is **pure**.

**Example:**

```
;; The polymorphic identity function.
;;
(plambda ((t type))
         (lambda ((x t)) x))
```

(proj *exp* $Dexp_1$...)

**Semantics:** *exp* must evaluate to a polymorphic value, as generated by plambda. The $Dexp_i$ are the actual parameters (or arguments) for the projection. The actuals are bound to the formal parameters specified in the definition of the polymorphic value. The number of actuals and the number of formals must be the same. Each $Dexp_i$ must be of the kind specified for the corresponding formal in the polymorphic value's definition. The body of the polymorphic value is evaluated with the formal parameters replaced by the actual parameters.

There is an important restriction on the way a polymorphic expression may be projected onto different arguments of kind region and effect: all the mutable region arguments must be mutually disjoint and must not intersect with free mutable region variables or constants used inside the type of the body of the polymorphic expression. This rule is called the *region anti-aliasing rule.* (Region descriptions which intersect are said to *alias*.)

Polymorphic subroutine values in the operator position of an application expression may be implicitly projected in most cases. (See page 41.)

**Type Information:** The type of a projection of a polymorphic value of type (poly (($d_1$ $Kexp_1$)...) *Texp*) is *Texp* with all occurrences of the $d_i$ replaced by their corresponding actual parameters.

**Effect Information:** The effect of a proj expression is pure.

**Example:**

```
;; Use the polymorphic identity function to get
;; a boolean version.
;;
(proj (plambda ((t type))
               (lambda ((x t)) x))
      bool)
```

$(exp\ exp_1 \ldots)$

**Semantics:** *exp* must evaluate to a polymorphic value, as generated by `plambda`. The polymorphic value is projected onto appropriate description values to produce a subroutine value, which is then applied to the $exp_i$. The projection and application are performed as described in the documentation for `proj` (page 40) and application (page 36).

The description values used as arguments to the projection are chosen so that the type of each of the $exp_i$ are subtypes of the types of the corresponding formal parameters of the resulting subroutine. An implicit projection is possible if the descriptions required as projection arguments are specified completely by the types of the actual parameters. If the types of the formal parameters do not utilize the `maxeff` or `runion` constructors, then the requisite projection arguments are specified completely, and will be used.

Projection arguments of kind `region` which are not otherwise implicitly specified by the type of the actual parameters are given the current value of the special description variable `default-region`. This variable is initially bound to the value `@=`, but may be rebound by the programmer using `plet` `pletrec`, or `plambda`.

**Type Information:** The type of an implicit projection and application of a polymorphic subroutine is the return type of the subroutine with all occurrences of the description variables bound by `plambda` replaced by the implicitly chosen projection arguments.

**Effect Information:** The effect of an implicit projection and application of a polymorphic value is computed by performing effect masking on the `maxeff` of the latent effect of the subroutine (with the $d_i$ replaced by the implicit projection arguments) and the effects of *exp* and the $exp_i$.

**Example:**

```
;; Invoke the polymorphic identity function with
;; a boolean argument, implicitly projecting to get a
;; boolean version.
;;
((plambda ((t type))
          (lambda ((x t)) x))
 #t)
```

(plet (($d_1$ $Dexp_1$)...) $exp_1$ $exp_2$...)

The body of the plet expression, $exp_1$ $exp_2$..., is treated as though (begin $exp_1$ $exp_2$...) is written.

The $d_i$ must all be distinct.

**Semantics:** plet provides a way of making type, effect, region and description function synonyms, or shorthand names, for complicated description expressions.

The value of a plet value expression is the value of its body. Whenever $d_i$ is encountered in the plet body, it is replaced by $Dexp_i$. A reference to a $d_i$ in $Dexp_j$ is taken to refer to a binding for $d_i$ in the surrounding (outer) scope. (See the description of pletrec for a discussion of recursive types.)

**Type Information:** The type of the plet expression is the type of its body with $Dexp_i$ substituted for $d_i$.

**Effect Information:** The effect of the plet expression is the effect of its body with $Dexp_i$ substituted for $d_i$.

**Example:**

```
;; The identity function on a complicated type.
;;
(plet ((t (ref (subr pure (bool) bool) @!)))
      (lambda ((a-subroutine t)) a-subroutine))
```

(**pletrec** (($d_1$ $Dexp_1$)...) $exp_1$ $exp_2$...)

The body of the **pletrec** expression, $exp_1$ $exp_2$..., is treated as though (**begin** $exp_1$ $exp_2$...) is written.

The $d_i$ must all be distinct.

**Semantics:** First, all of the $d_i$ are read and these names are made available. Then, the $Dexp_i$ are successively evaluated and bound to the corresponding $d_i$. This order is important because it allows any of the $Dexp_j$ to refer to some $d_i$, thus providing for mutual recursion. This process is subject to the following restriction: all description variables that can be recursively reached, in the graph of description variable usage, from an expression $Dexp_i$ must be defined *before* the processing of the binding for $d_i$, unless their kinds are type. Typically, the $Dexp_j$ are recursive type descriptions and so there is no problem. An error is signalled if any of the $d_i$ is defined as itself.

The value of a **pletrec** special form is its evaluated body with the (possibly recursive) definitions of the $d_i$ substituted for the $d_i$.

**Type Information:** *The type of a* **pletrec** *value expression* is the type of its body with all occurrences of $d_i$ replaced by $Dexp_i$.

**Effect Information:** The effect of a **pletrec** construct is the effect of its body with $Dexp_i$ substituted for $d_i$.

**Example:**

```
;; A contrived example.
;;
(pletrec ((t1 (subr pure (bool) t2))
          (t2 (subr pure () t1)))
         (lambda ((x t1))
                 ((x #t))))
```

(if *exp*$_0$ *exp*$_1$ *exp*$_2$)

**Semantics:** The expression *exp*$_0$ must be of type bool. If *exp*$_0$ evaluates to #t, then the value of the if expression is the value of *exp*$_1$, otherwise the value of the if is the value of *exp*$_2$. The type of one of the arms of the if expression must be a subtype of the other.

**Type Information:** The type of an if expression is the maximum of the types of *exp*$_1$ and *exp*$_2$.

**Effect Information:** The effect of an if statement is the maxeff of the effect of the three expressions *exp*$_0$, *exp*$_1$, and *exp*$_2$.

**Examples:**

```
;; The short-circuit "and".
;;
(if x y #f)

;; The "not" function.
;;
(lambda (x) (if x #f #t))
```

(**set!** *var exp*)

**Semantics:** **set!** is the variable assignment operator. First, *exp* is evaluated. Then, the resulting value is placed in the location denoted by *var*. The value of a **set!** expression is **#u**.

It is a static effect error for *var* to be in the region **@***. Furthermore, if *var* is of type *Texp*, then the type of *exp* must be a subtype of *Texp*.

**Type Information:** The type of the **set!** expression is **unit**.

**Effect Information:** The effect of an assignment expression is (**write** *r*), where *r* is the region *var* is in.

**Examples:**

```
;; Mutate the boolean variable x to #t.
;;
(set! x #t))

;; Set a boolean flag.
;;
(set! flag (and? a b))
```

In addition to the above language constructs, the kernel provides three subroutines which may be used with `ref` types. They are:

```
new  : (poly ((r region))
         (poly ((t type))
               (subr (alloc r) (t) (ref t r))))

get  : (poly ((r region))
         (poly ((t type))
               (subr (read r) ((ref t r)) t)))

set  : (poly ((r region))
         (poly ((t type))
               (subr (write r) ((ref t r) t) unit)))
```

`new` is used to allocate a new location in a particular region and initialize the location to some value. `((proj new @!) #t)` returns a reference to a newly allocated location of type `(ref bool @!)` which contains the initial value `#t`.

`get` is used to *dereference* a value of `ref` type, *i.e.* to return the value currently stored in the location indicated by the reference value.

`set` is used to replace the value stored in the location denoted by a `ref` type with a new value.

# Chapter 3

# The FX Standard Types

This chapter describes the ***standard types*** that are provided by every *FX* implementation. These types fill out the framework introduced by the *FX* Kernel with a set of useful types and subroutines. We present the *FX* standard types in order of increasing complexity.

For each data type, we give a brief overview of its purpose, the syntax of literals, and a list of the operations provided.

Many of the subroutines described in this chapter are abstracted over multiple descriptions (see for instance, the `caar` subroutine which is abstracted over one region and three types). As a general rule, region parameters are abstracted over first in the definition of a standard subroutine, followed by other descriptions. This currying of description abstraction allows a programmer to specify the region for a standard subroutine with a `proj` expression and to omit the `proj` for the other description parameters. The other description parameters will be computed by implicit projection (See page 41 for a description of implicit projection.)

Standard subroutines are generally abstracted over only one region parameter. If the values to be operated upon are in different regions, the user has to pass the `runion` of those regions as the argument. If the operations were abstracted over multiple regions, then the rule that disallows passing the same region as an argument to two region parameters would prevent the use of the operation on values in the same region.

Unless otherwise stated, there is no type inclusion within these data types, and literals are always immutable.

## 3.1 Void

The type void is the type of certain non-terminating computations. For example, it is the return type of the error function which is described in Chapter 5.

### Description

The void constant type is the empty type, *i.e.* there are no values of type void; its kind is type.

The type void is a subtype of all types.

### Literals

There are no literals of type void.

### Operations

There are no operations for the type void.

This type is rarely used; one contrived example of its use is:

```
(letrec ((black-hole (lambda ()
                             (the pure void
                                  (black-hole)))))
        (black-hole))
```

## 3.2 Unit

The unit constant data type is the type of computations that are only used to perform side-effects. It is already defined in the *FX* Kernel (cf. previous chapter). Its kind is type.

### Literals

There is one value of type unit, namely #u.

### Operations

There are no operations for the type unit.

The type unit is generally used as the type of computations which return no useful information.

```
;; The variable foo is located in region @bar.
;;
(let ((mutate (the (subr (write @bar) () unit)
                   (lambda ()
                           (set! foo #t)))))
     (mutate))
```

## 3.3 Booleans

The `bool` constant data type denotes the set of immutable boolean values. It is already defined in the *FX* Kernel (cf. previous chapter).

### Literals

There are two boolean literals, namely `#t` (for the *true* boolean) and `#f` (for the *false* boolean).

### Operations

The *FX* implementation provides the classical boolean operators:

| | | |
|---|---|---|
| `equiv?` | : | `(subr pure (bool bool) bool)` |
| `and?` | : | `(subr pure (bool bool) bool)` |
| `or?` | : | `(subr pure (bool bool) bool)` |
| `not?` | : | `(subr pure (bool) bool)` |

The more classical and and or special forms, which perform short-circuit evaluations, are defined in the next chapter. Note also that `equiv?`, which is the equality function on booleans, could be easily defined with the other functions; `equiv?` is provided as a convenience to the programmer.

## 3.4 Integers

The `int` constant data type denotes the set of immutable integers. The kind of `int` is `type`.

### Literals

The *FX* `int` data type supports four distinct bases for integer literals. The distinction is indicated by a prefix, namely `#b` (binary), `#o` (octal), `#d` (decimal) and `#x` (hexadecimal). If no prefix is supplied, `#d` is assumed.

An integer literal is formed by an optional prefix, an optional sign (+ is assumed if omitted), and a non-empty succession of digits that are defined in the given base. The precision of integer values is unspecified.

### Operations

The integer operations with their types are:

```
= :  (subr pure (int int) bool)
< :  (subr pure (int int) bool)
> :  (subr pure (int int) bool)
<=:  (subr pure (int int) bool)
>=:  (subr pure (int int) bool)
```

These are the five standard comparison functions on integers.

```
+ :  (subr pure (int int) int)
* :  (subr pure (int int) int)
- :  (subr pure (int int) int)
/ :  (subr pure (int int) int)
```

These are the four standard arithmetic operations on integers. A dynamic error is signalled in case of division by zero or overflow.

```
remainder :  (subr pure (int int) int)
modulo    :  (subr pure (int int) int)
abs       :  (subr pure (int) int)
```

The first two functions implement number-theoretic integer division; the functions `remainder` and `modulo` differ on negative arguments (`remainder` has always the sign of the dividend). The `abs` function erases the sign of its argument.

## 3.5 Floating-point numbers

The `float` constant data type denotes the set of immutable real numbers. The kind of `float` is `type`.

## Literals

The *FX* `float` data type supports the standard FORTRAN-inspired syntax for literals. These are typical examples of floating-point number literals:

| | |
|---|---|
| `+0.66666` | *a float approximation of 2/3* |
| `6.66e-1` | *a less accurate one* |
| `-6.66E-1` | *its opposite value* |
| `0.0` | *the floating-point number zero* |

A float literal is formed by an optional sign, a non-empty succession of decimal digits, a decimal point, a non-empty succesion of decimal digits and an optional exponent denoted by the letter "E" or "e", an optional sign and a sequence of decimal digits. The precision of floating point values is unspecified; this means that truncation may occur if the number of significant digits is too large.

## Operations

The floating-point operations with their types are:

```
fl= :  (subr pure (float float) bool)
fl< :  (subr pure (float float) bool)
fl> :  (subr pure (float float) bool)
fl<=:  (subr pure (float float) bool)
fl>=:  (subr pure (float float) bool)
```

These are the five standard comparison functions on floats.

```
fl+ :  (subr pure (float float) float)
fl* :  (subr pure (float float) float)
fl- :  (subr pure (float float) float)
fl/ :  (subr pure (float float) float)
```

These are the four standard arithmetic operations on floats. A dynamic error is signalled in case of division by zero, overflow or underflow.

```
flabs :  (subr pure (float) float)
```

This function erases the sign of its argument.

```
exp :  (subr pure (float) float)
log :  (subr pure (float) float)
sin :  (subr pure (float) float)
cos :  (subr pure (float) float)
tan :  (subr pure (float) float)
asin:  (subr pure (float) float)
acos:  (subr pure (float) float)
atan:  (subr pure (float) float)
sqrt:  (subr pure (float) float)
```

These are the basic arithmetic operations on floats.

```
floor     :  (subr pure (float) int)
ceiling   :  (subr pure (float) int)
truncate  :  (subr pure (float) int)
round     :  (subr pure (float) int)

int->float:  (subr pure (int) float)
```

We provide the classical conversion functions from integers to floats, and vice-versa.

## 3.6 Characters

The `char` constant type denotes the set of immutable characters. The kind of `char` is `type`.

### Literals

Character literals are represented with the `#\`*character* or `#\`*identifier* notation and must be followed by a delimiter. For instance, `#\a` is the lower case "a" character, while `#\Z` is the upper case letter "z"; `#\newline` denotes the NewLine character.

The list of allowed identifiers must include:

```
backspace  newline  page
space      tab
```

The case used in the character identifiers is irrelevant; `#\newline` is equivalent to `#\NewLine` for example.

## Operations

The operations on characters are either case sensitive or case insensitive. The latter option is indicated by a `-ci` suffix in the operation name.

There is a total ordering on characters, which is compatible with the ASCII standard on lower-case letters, upper-case letters and digits (without any interleaving between letters and digits).

The operations defined on characters with their types are:

```
char=?  :  (subr pure (char char) bool)
char<?  :  (subr pure (char char) bool)
char>?  :  (subr pure (char char) bool)
char<=? :  (subr pure (char char) bool)
char>=? :  (subr pure (char char) bool)
```

These are the five boolean comparison functions on characters.

```
char-ci=?  :  (subr pure (char char) bool)
char-ci<?  :  (subr pure (char char) bool)
char-ci>?  :  (subr pure (char char) bool)
char-ci<=? :  (subr pure (char char) bool)
char-ci>=? :  (subr pure (char char) bool)
```

These five comparison functions treat upper- and lower-characters as the same.

```
char-alphabetic?  :  (subr pure (char) bool)
char-numeric?     :  (subr pure (char) bool)
char-whitespace?  :  (subr pure (char) bool)
```

A character is alphabetic if its lower-case equivalent is between `#\a` and `#\z`. It is numeric if it is between `#\0` and `#\9`.

```
char-lower-case?  :  (subr pure (char) bool)
char-upper-case?  :  (subr pure (char) bool)
char-upcase       :  (subr pure (char) char)
char-downcase     :  (subr pure (char) char)
```

The two boolean-valued subroutines test the case of a character. The two last subroutines map a character to the corresponding case; non-alphabetic characters remain unchanged.

```
char->int  :  (subr pure (char) int)
int->char  :  (subr pure (int) char)
```

These two subroutines convert between characters and their positions in the ordering mentioned above.

## 3.7 Strings

The `string` type constructor is used to denote the set of zero-indexed sequences of characters. Once created, a string is of constant length.

### Description

The type of a string located in a region *Rexp* is:

`(string` ***Rexp***`)  ::  type`

and the kind of `string` is:

`string  ::  (dfunc (region) type)`

A type (`string` ***Rexp***$_1$) is a subtype of (`string` ***Rexp***$_2$) iff ***Rexp***$_1$ is a subregion of ***Rexp***$_2$.

### Literals

A string literal is represented by a double-quote ("), a sequence of characters (where \ is the escape character for itself and the double-quote character), and an ending double-quote.

### Operations

The operations on strings with their types are:

```
make-string   :  (poly ((r region))
                       (subr (alloc r)
                             (int char) (string r)))
string-length :  (poly ((r region))
                       (subr pure ((string r)) int))
```

The **make-string** function creates a string in the region **r** of the length given by the first argument and fills it with the second argument. The latent effect of **string-length** is pure since the length of a string, which is constant, can be obtained without looking at (*i.e.* have a **read** effect on) the string value.

```
string-ref :  (poly ((r region))
                    (subr (read r)
                          ((string r) int) char))
string-set!  :  (poly ((r region))
                      (subr (write r)
                            ((string r) int char) unit))
string-fill!  :  (poly ((r region))
                       (subr (write r)
                             ((string r) char) unit))
substring-fill!  :
  (poly ((r region))
        (subr (write r)
              ((string r) int int char) unit))
```

The function **string-ref** returns the *n*-th character in a string where *n* is the second argument. The subroutine **string-set!** modifies its argument at the given index. The subroutine **string-fill!** fills its argument with the given character. The last subroutine (**substring-fill!**) allows one to fill a part of a string by the character given as the last argument; the first **int** gives the beginning index and the second is one greater than the index of the last position in the substring. It is a dynamic error to try to access out-of-bounds elements of strings. The mutating subroutines return **#u**.

```
string=?,
string<?,
string>?,
string<=?,
string>=?,
string-ci=?,
string-ci<?,
string-ci>?,
string-ci<=?,
string-ci>=?  :
  (poly ((r region))
     (subr (read r)
           ((string r) (string r)) bool))
```

These are the lexicographic comparison functions on strings; the case-insensitive ones have a `-ci` suffix.

```
substring :
  (poly ((r1 region))
     (subr (read r1)
           ((string r1) int int)
           (poly ((r2 region))
                 (subr (alloc r2) () (string r2)))))
string-append :
  (poly ((r1 region))
     (subr (read r1)
           ((string r1) (string r1))
           (poly ((r2 region))
                 (subr (alloc r2) () (string r2)))))
string-copy :
  (poly ((r1 region))
     (subr (read r1)
           ((string r1))
           (poly ((r2 region))
                 (subr (alloc r2) () (string r2)))))
```

These functions create newly allocated strings from their argument(s).

The `substring` arguments must specify valid index ranges. More precisely, it is a dynamic error if the first integer argument (which is the index into the string argument `s`, located in `r`, of the first character to be included in the substring) is not between zero and the length of the string minus one, inclusive, if the second integer argument (which is one larger than the index into `s` of the last character to be included in the substring) is not between zero and the length of the string, inclusive, and if the first integer argument is not less than or equal to the second. If the two integer arguments are equal, then the substring returned is the empty string (`""`).

For example, consider

```
      ((proj ((proj substring @from) s start end) @to))
```

`start` is the index in the string `s` of the first character of the result; it should be less than or equal to end which is the last (non-included) character index in `s`. The following table shows some sample results for different values of `s`, `start`, and `end`:

| `s` | `start` | `end` | `result` |
|---|---|---|---|
| `"boondoggle"` | 4 | 7 | `"dog"` |
| `"foobar"` | 3 | 3 | `""` |
| `"insipid"` | 4 | 7 | `"pid"` |
| `"yuk"` | 0 | 1 | `"y"` |

The `string-append` function implements the concatenation of its arguments.

`string-copy` yields a fresh copy of its argument.

## 3.8 Symbols

The `symbol` constant data type is used to denote the set of immutable values whose name is the only important characteristic. Its kind is `type`.

### Literals

A symbol literal is represented with the `quote` special form.

(quote *id*)
or
'*id*

*id* is any identifier (see the Conventions section).
'*id* is equivalent to (quote *id*).

**Semantics:** A quoted identifier evaluates to the symbol whose name is the upper-case equivalent of the identifier.

**Type Information:** (quote *id*) is of type symbol.

**Effect Information:** A quote expression is pure.

**Example:**

```
;; The following expression returns #t.
;;
(symbol=?  (quote Foo) 'fOo)
```

### Operations

We provide conversion functions on symbols to and from strings:

```
symbol->string :
  (poly ((r region))
        (subr (alloc r) (symbol) (string r)))
string->symbol :
  (poly ((r region))
        (subr (read r) ((string r)) symbol))
```

However, the most important characteristic of symbols is that they are treated in a very special way by the *FX* interpreter or compiler; they are *interned*. For instance, if the symbol `'foo` is used in two different places in a program, they will in fact refer to the *same* (physical) value. To detect whether two symbol values are the same in this very precise sense, we provide the `symbol=?` function which tests for the physical equality (*i.e.*, the identity) of two symbols:

```
symbol=?  :  (subr pure (symbol symbol) bool)
```

The `hash` function computes a hash code from its argument.

```
hash :  (subr pure (symbol) int)
```

## 3.9 References

The `ref` type constructor is used to denote the set of values that are references to other values. It is already defined in the *FX* Kernel (cf. previous chapter).

### Description

The type of a reference, located in a region *Rexp*, to a value of type *Texp* is:

(`ref` *Texp Rexp*) :: `type`

and the kind of `ref` is:

`ref` :: (`dfunc` (`type region`) `type`)

A type (`ref` $Texp_1$ $Rexp_1$) is a subtype of (`ref` $Texp_2$ $Rexp_2$) iff $Rexp_1$ is a subregion of $Rexp_2$ and $Texp_1$ is interconvertible to $Texp_2$, or $Rexp_1 = Rexp_2$ = `@=` and $Texp_1$ is a subtype of $Texp_2$.

### Literals

There are no literals for `ref` values.

### Operations

There are three operations on reference values, which have already been introduced in the Chapter 2.

```
new :  (poly ((r region))
         (poly ((t type))
               (subr (alloc r) (t) (ref t r))))
get :  (poly ((r region))
         (poly ((t type))
               (subr (read r) ((ref t r)) t)))
set :  (poly ((r region))
         (poly ((t type))
               (subr (write r) ((ref t r)) unit)))
```

It is a static effect error to apply `set` to a `ref` value in the region `@=`.

## 3.10 Uniqueofs

The `uniqueof` type constructor is used to denote sets of values in which each element is distinguishable.

### Description

The type of a unique value of type *Texp* is

(`uniqueof` *Texp*) :: `type`

and the kind of `uniqueof` is:

`uniqueof` :: (`dfunc` (`type`) `type`)

A type (`uniqueof` $Texp_1$) is a subtype of (`uniqueof` $Texp_2$) iff $Texp_1$ is a subtype of $Texp_2$.

### Literals

There are no literals for uniqueof values.

## Operations

There are three operations on uniqueof values:

```
unique :  (poly ((t type))
                (subr (alloc @uniqueof) (t) (uniqueof t)))
```

The unique subroutine creates a unique value from a value of type t. The alloc effect on @uniqueof is used to ensure that no memoization will be performed on calls to unique.

```
value :  (poly ((t type))
               (subr pure (uniqueof t) t))
```

The value subroutine returns the embedded value corresponding to a unique value.

```
eq?  :  (poly ((t1 type) (t2 type))
              (subr pure ((uniqueof t1) (uniqueof t2)) bool))
```

The boolean-valued subroutine eq? tests whether two unique values were created by the same call to the unique function.

## 3.11 Pairs

The notion of pair in *FX* is the same as the standard Lisp one.

### Description

The type of a pair, located in a region *Rexp*, whose first element (CAR) is of type $Texp_1$ and its second (CDR) is of type $Texp_2$ is

(pairof $Texp_1$ $Texp_2$ *Rexp*) :: type

and the kind of pairof is:

pairof :: (dfunc (type type region) type)

A type (pairof $Texp_{11}$ $Texp_{12}$ $Rexp_1$) is a subtype of (pairof $Texp_{21}$ $Texp_{22}$ $Rexp_2$) iff $Rexp_1$ is a subregion of $Rexp_2$, $Texp_{11}$ is interconvertible to $Texp_{21}$ and $Texp_{12}$ is interconvertible to $Texp_{22}$, or $Rexp_1 = Rexp_2 =$ @=, $Texp_{11}$ is a subtype of $Texp_{21}$ and $Texp_{12}$ is a subtype of $Texp_{22}$.

## Literals

There are no literals for pair values.

## Operations

We provide the standard set of operations on pairs.

```
null?  :  (poly ((r region))
            (poly ((t1 type) (t2 type))
                  (subr pure ((pairof t1 t2 r)) bool)))
```

The function `null?` tests whether a pair is actually null *i.e.* whether it is equal to `()` (see the next section for a description of the `null` type).

```
cons :
  (poly ((r region))
    (poly ((t1 type) (t2 type))
          (subr (alloc r) (t1 t2) (pairof t1 t2 r))))
car :
  (poly ((r region))
    (poly ((t1 type) (t2 type))
          (subr (read r) ((pairof t1 t2 r)) t1)))
cdr :
  (poly ((r region))
    (poly ((t1 type) (t2 type))
          (subr (read r) ((pairof t1 t2 r)) t2)))
caar :
  (poly ((r region))
    (poly ((t1 type) (t2 type) (t3 type))
          (subr (read r)
                ((pairof (pairof t1 t2 r) t3 r)) t1)))
cadar :
  (poly ((r region))
    (poly ((t1 type) (t2 type) (t3 type) (t4 type))
          (subr (read r)
                ((pairof (pairof t1
                                 (pairof t2 t3 r)
                                 r)
                         t4
                         r))
```

```
                 t2)))
caaaar :   idem, con variatione
```

The standard CONS, CAR, CDR and C{A|D}$^+$R operations are defined in *FX* (up to four A's or D's in C...R). We did not give the types of all the C...R constructs since they are all similar.

```
set-car!  :
  (poly ((r region))
    (poly ((t1 type) (t2 type))
          (subr (write r) ((pairof t1 t2 r) t1) unit)))
set-cdr!  :
  (poly ((r region))
    (poly ((t1 type) (t2 type))
          (subr (write r) ((pairof t1 t2 r) t2) unit)))
```

These are the standard Lisp forms for mutation of pairs. These subroutines cannot be used with pairs of pairs that span more than one region. The subtyping rule for pairs does not allow the programmer to pass a pair to the subroutine if the pair is not in the precise region expected. The programmer may supply his own subroutines to handle such complicated structures.

## 3.12 Null

The `null` data type is provided as the type of the list with no elements.

### Description

The kind of `null` is type.

The type `null` is a subtype of every type of the form (`pairof` $Texp_1$ $Texp_2$ $Rexp$). This subtyping rule is safe since there is no mutable value of type `null`.

### Literals

The only value of type `null` is the literal `()`.

### Operations

There are no operations which explicitly use `null` as the type of an argument or as the type of a returned value.

Since `null` is a subtype of any pair type, some operations on pairs may be applied to `()`. It is a dynamic error to apply pair access operations such as `car` or `cdr` on `()`. A dynamic error is signalled if **`set-car!`** or **`set-cdr!`** is applied to `()`.

## 3.13 Lists

The type constructor **`listof`** is used to denote the set of homogeneous lists of a given type, allocated in a given region.

### Description

We define **`listof`** in terms of **`pairof`**:

```
(dlambda ((t type) (r region))
         (dletrec ((listof-t-in-r (pairof t listof-t-in-r r)))
                  listof-t-in-r))
```

The type of a list defined in the region *Rexp* with elements of type *Texp* is then:

**(`listof` *Texp Rexp*) :: type**

and the kind of **`listof`** is:

**`listof` :: (dfunc (type region) type)**

The seemingly infinite recursive definition of **`listof`** does not prevent us from producing values of type **`listof`** since **`null`** is a subtype of any (**`pairof`** *Texp*$_1$ *Texp*$_2$ *Rexp*) and, then, of any (**`listof`** *Texp Rexp*). Therefore, `()` is the terminator of every list. Moreover, `()` is used to represent the empty list.

### Literals

There are no list literals for list values, other than `()`.

### Operations

We provide the classical operations on lists. Remember that the operations defined for pairs apply equally here. In particular, use the **`null?`** subroutine to determine if a list is empty.

```
list :  (poly ((r region))
          (poly ((t type))
                (vsubr (alloc r) t (listof t r))))
```

This function takes a *variable* number of arguments of type t; its type is built with the vsubr type constructor, which is defined in the next section.

```
length :  (poly ((r region))
            (poly ((t type))
                  (subr (read r) ((listof t r)) int)))
```

The length of a list may be changed by the program by using `set-cdr!`. `length` has a read effect because it has to take the cdr of each pair of the list until it gets to the end.

```
append :  (poly ((r region))
            (poly ((t type))
                  (subr (maxeff (read r) (alloc r))
                        ((listof t r) (listof t r))
                        (listof t r))))
reverse :  (poly ((r1 region))
             (poly ((t type))
                   (subr (read r1)
                         ((listof t r1))
                         (poly ((r2 region))
                            (subr (alloc r2)
                                  () (listof t r2))))))
list-tail :  (poly ((r region))
               (poly ((t type))
                     (subr (read r)
                           ((listof t r) int) (listof t r))))
list-ref :  (poly ((r region))
              (poly ((t type))
                    (subr (read r)
                          ((listof t r) int) t)))
```

The function call (list-tail l k) returns the sublist of l after omitting the first k elements. The function call (list-ref l k) yields the k-th element of the list l (the first element being the zero-th).

```
memq :
  (poly ((r region))
    (poly ((t type))
          (subr (read r)
                ((uniqueof t) (listof (uniqueof t) r))
                (listof (uniqueof t) r))))
assq :
  (poly ((r region))
    (poly ((t1 type) (t2 type))
          (subr (read r)
                ((uniqueof t1)
                 (listof (pairof (uniqueof t1) t2 r) r))
                (pairof (uniqueof t1) t2 r))))
```

Since the memq and assq functions traditionally use the eq? predicate, they cannot be abstracted over any sort of *FX* types, but are limited to uniqueofs. Note that, contrary to Scheme, these functions return () whenever the uniqueof argument does not match any element of the list passed as a second argument.

```
member :
  (poly ((r region))
    (poly ((t type) (e effect))
          (subr (maxeff (read r) e)
                ((subr e (t t) bool)
                 t
                 (listof t r))
                (listof t r))))
assoc :
  (poly ((r1 region))
    (poly ((t1 type) (t2 type) (e effect))
          (subr (maxeff (read r) e)
                ((subr e (t1 t2) bool)
                 t1
                 (listof (pairof t1 t2 r) r))
                (pairof t1 t2 r))))
```

Contrary to the Lisp usage, the **member** and **assoc** functions have to be provided with a comparison predicate (there is no way to provide in *FX* a well-typed function equivalent to the general Lisp equal? predicate).

```
string->list :
  (poly ((r region))
        (subr (maxeff (read r) (alloc r))
              ((string r)) (listof r char)))
list->string :
  (poly ((r region))
        (subr (maxeff (read r) (alloc r))
              ((listof r char)) (string r)))
```

These functions enable the conversion of lists of characters to and from strings.

The fairly standard control structures provided by Lisp on lists are fully provided in the *FX* implementation.

```
map :
  (poly ((r region))
    (poly ((t1 type) (t2 type) (e effect))
          (subr (maxeff e (read r) (alloc r))
                ((subr e (t1) t2) (listof t1 r))
                (listof t2 r))))
for-each :
  (poly ((r region))
    (poly ((t1 type) (t2 type) (e effect))
          (subr (maxeff e (read r))
                ((subr e (t1) t2) (listof t1 r))
                unit)))
reduce :
  (poly ((r region))
    (poly ((t type) (e effect))
          (subr (maxeff e (read r))
                ((subr e (t t) t) (listof t r) t)
                t)))
```

where the functions `map` and `for-each` are performed from left to right and where the reduction done by `reduce` is right-associative, *e.g.*:

```
(reduce + (list 1 2 3) 0) = (+ 1 (+ 2 (+ 3 0)))
```

## 3.14 Vsubrs

MACLISP "lexprs" and CommonLISP **&rest** arguments are well-known techniques to allow the definition of subroutines which accept a variable number of arguments. This feature is provided in *FX* with the type constructor vsubr (for "variable-subr").

### Description

The type of a subroutine that accepts a variable number of arguments of type *Texp* (a list of which is bound to the sole formal parameter), returns a value of type $Texp_{rtn}$, and has a latent effect *Eexp* is:

(vsubr *Eexp Texp* $Texp_{rtn}$) :: type

The kind of vsubr is

```
vsubr :: (dfunc (effect type type) type)
```

A function of type (vsubr $Eexp_1$ $Texp_1$ $Texp_{rtn1}$) is a subtype of (vsubr $Eexp_2$ $Texp_2$ $Texp_{rtn2}$) iff $Eexp_1$ is a subeffect of $Eexp_2$, $Texp_2$ is a subtype of $Texp_1$ and $Texp_{rtn1}$ is a subtype of $Texp_{rtn2}$.

### Literals

There are no literals for vsubr values.

### Operations

```
apply :
  (poly ((r region))
    (poly ((t1 type) (t2 type) (e effect))
          (subr (maxeff e (read r))
                ((vsubr e t1 t2)
                 (listof t1 r))
                t2)))
```

There is one special form to build vsubr expressions, namely vlambda, and one to use them, namely Variable-length Application.

(**vlambda** (*var Texp* [*Rexp*]) $exp_1$ $exp_2$ ... $exp_m$)

**Semantics:** See page 34 for the detailed semantics of the basic `lambda` construct (*i.e.*, without the special binding).

The subroutine takes zero or more arguments, each of which must have a type compatible with *Texp*. When the function is applied to these *n* arguments, they are gathered in a list of type (`listof` *Texp* `@=`); this list is then bound to *var* allocated in the region *Rexp* (which defaults to `@=`).

**Type Information:** If the type of the body of the `vlambda` is $Texp_{body}$, then the type of the subroutine value created is:

(`vsubr` *Eexp Texp* $Texp_{body}$)

where *Eexp* is the *latent effect* of the subroutine. *Eexp* is computed by performing effect masking on the `maxeff` of (`alloc` *Rexp*) and the effect of the $exp_i$.

**Effect Information:** The effect of a `vlambda` expression is pure.

**Example:**

```
;; A pre-projected (on @=) implementation of list.
;;
(plambda ((t type))
        (vlambda (l t)
                l))
```

$(exp\ exp_1 \ldots)$

**Semantics:** The expression *exp* must evaluate to a vsubr value. The $exp_i$ are the actual parameters, or arguments to the subroutine. The expressions are all evaluated from left to right (*exp* first). The evaluated arguments are gathered into a list (allocated in @=). Then, the list is bound to the variable which is the formal parameter specified in the definition of the vsubr value.

The body of the subroutine is then evaluated with the formal so bound. The value of the variable-length application form is the value obtained by thus evaluating the subroutine body.

**Type Information:** Each of the $exp_i$ must have a subtype of the type given with the corresponding formal parameter; if *exp* evaluates to a subroutine value of type (vsubr *Eexp Texp* $Texp_{body}$), then $exp_i$ must have a subtype of type *Texp*.

The type of the application expression is the return type, $Texp_{body}$, of the subroutine.

**Effect Information:** The effect of the application expression is computed by performing effect masking on the `maxeff` of the latent effect of the subroutine and the effects of *exp* and the $exp_i$.

**Example:**

```
;; Vlambda as a list constructor.
;;
((vlambda (l int) l) 1 2 3 4)
```

## 3.15 Promises

The `promise` data type is used to describe *delayed* values in the sense of Scheme. A delayed value denotes the "promise" of a future evaluation of a given expression; the precise moment when this suspended expression is evaluated is controlled by the user. This type constructor and its operations can be used to create potentially infinite data structures.

### Description

The type of a delayed value is:

(`promise` *Eexp* *Texp*) :: `type`

where *Eexp* is the effect of evaluating the delayed expression and *Texp* is the type of the delayed expression. The kind of `promise` is:

`promise` :: (`dfunc` (`effect` `type`) `type`)

A promise value of type (`promise` $Eexp_1$ $Texp_1$) is a subtype of (`promise` $Eexp_2$ $Texp_2$) iff $Eexp_1$ is a subeffect of $Eexp_2$ and $Texp_1$ is a subtype of $Texp_2$.

### Literals

There are no literals for promise values.

### Operations

There is only one special form and one subroutine that deal with promise expressions: `delay` creates a delayed expression and `force` evaluates its argument and returns the resulting value.

(delay *exp*)

**Semantics:** This special form creates a delayed value from its (unevaluated) argument. The result of the delay expression is the newly allocated delayed value.

**Type Information:** The type of the delay expression is constructed with the type and (latent) effect, *Texp* and *Eexp*, of *exp*:

(promise *Eexp Texp*)

**Effect Information:** If the effect of *exp* is pure, then the effect of (delay *exp*) is also pure. Otherwise, the delay expression has effect (alloc @promise).

**Example:**

```
;; A generator of streams
;;
(pletrec ((int-stream (pairof int
                              (promise (alloc (runion @s @promise))
                                       int-stream)
                              @s)))
  (letrec ((next (lambda ((n int))
                   (the (alloc (runion @s @promise)) int-stream
                        ((proj cons @s)
                         n
                         (delay (next (+ n 1))))))))
    (next 0)))
```

The `force` subroutine, which evaluates a delayed expression, has type:

```
force:
  (poly ((e effect) (t type))
        (subr e ((promise e t)) t))
```

Note that forced expressions are memoized; forcing a delayed value twice is equivalent to a single `force` operation.

## 3.16 Vectors

The notion of integer-indexed, homogeneous data structure is provided in *FX* by the `vectorof` type. In *FX*, vectors (sometimes called arrays in other languages) are indexed starting at zero and, once created, are of constant length.

### Description

A vector, which is located in a region *Rexp* and contains elements of type *Texp*, has type:

(`vectorof` *Texp Rexp*) :: `type`

The kind of `vectorof` is:

`vectorof` :: (`dfunc` (`type region`) `type`)

A type (`vectorof` $Texp_1$ $Rexp_1$) is a subtype of (`vectorof` $Texp_2$ $Rexp_2$) iff $Rexp_1$ is a subregion of $Rexp_2$ and $Texp_1$ is interconvertible to $Texp_2$, or $Rexp_1 = Rexp_2 =$ `@=` and $Texp_1$ is a subtype of $Texp_2$.

### Literals

There are no literals for vector values.

### Operations

The allowed operations on vectors are given below:

```
make-vector :
  (poly ((r region))
    (poly ((t type))
            (subr (alloc r) (int t) (vectorof t r))))
vector :
  (poly ((r region))
    (poly ((t type))
            (vsubr (alloc r) t (vectorof t r))))
vector-length :
  (poly ((r region))
    (poly ((t type))
            (subr pure ((vectorof t r)) int)))
vector-ref :
  (poly ((r region))
    (poly ((t type))
            (subr (read r) ((vectorof t r) int) t)))
```

The make-vector function creates a new vector the number of elements of which is given by its first argument and the initial content by the second. The vector function takes a variable number of arguments and creates a new vector with them as initial values.

The latent effect of vector-length is pure since the length of a vector, which is constant, can be obtained without looking at (*i.e.* have a read effect on) the vector value.

The vector-ref function yields the value associated with an index in a vector; it is a dynamic error if the index is not valid, *i.e.* between zero and the length of the vector minus one inclusive.

```
vector-set! :
  (poly ((r region))
    (poly ((t type))
            (subr (write r)
                  ((vectorof t r) int t) unit)))
vector-fill! :
  (poly ((r region))
    (poly ((t type))
            (subr (write r)
                  ((vectorof t r) t) unit)))
```

The vector-set! function modifies its vector argument by setting the value of the third argument at the given index (second argument) . The vector-fill! function mutates its vector argument by filling it with the second argument. These two functions return #u.

```
vector->list :
  (poly ((r region))
    (poly ((t type))
          (subr (maxeff (read r) (alloc r))
                ((vectorof t r)) (listof t r))))
list->vector :
  (poly ((r region))
    (poly ((t type))
          (subr (maxeff (read r) (alloc r))
                ((listof t r)) (vectorof t r))))
```

These functions convert vectors to and from lists.

## 3.17 Records

The notion of heterogeneous data structures with named fields is introduced in the *FX* language via the **recordof** type constructor. There are three special forms for manipulating recordof values: **record**, **select** and **record-set!**.

$(\texttt{recordof}\ ((n_1\ Texp_1) \ldots (n_m\ Texp_m))\ Rexp)$

**Semantics:** This is the type of a record (*i.e.*, an ordered aggregate data structure containing zero or more fields) defined in region *Rexp* with fields $(n_1\ Texp_1), \ldots (n_m\ Texp_m)$.

Each field has a name (an identifier) and a type. The order of the fields within the recordof type is relevant and the field names must be distinct.

A record type (recordof $((n_1\ Texp_{11}) \ldots (n_m\ Texp_{1m}))\ Rexp_1$) is a subtype of (recordof $((n_1\ Texp_{21}) \ldots (n_q\ Texp_{2q}))\ Rexp_2$) iff $Rexp_1$ is a subregion of $Rexp_2$, $m \geq q$, and all the $Texp_{1i}$ are interconvertible to $Texp_{2i}$, or

1. $Rexp_1 = Rexp_2 =$ **@=**
2. $m \geq q$
3. for each field name $n_i$, $Texp_{1i}$ is a subtype of $Texp_{2i}$

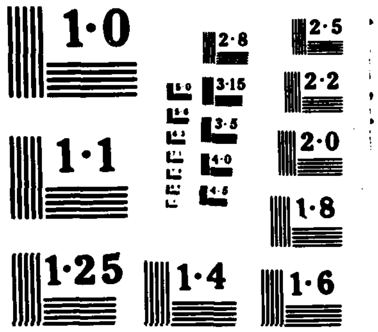
There are no literals for record values.

**Kind Information:** The kind of a recordof expression is type.

**Example:**

```
;; The type of a person record.
;;
(recordof ((name (string @=))
           (address (string @=))
           (phone-number (string @=)))
          @Persons)
```

AD-A187 031 FX-87 REFERENCE MANUAL EDITION 10(U) MASSACHUSETTS INST 2/2
OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE
D K GIFFORD ET AL SEP 87 MIT/LCS/TR-407
UNCLASSIFIED N00014-83-K-0125 F/G 12/5 NL


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

(**record** (($n_1$ $exp_1$) ...($n_m$ $exp_m$)) [$Rexp$])

The field names $n_i$ must all be distinct.

**Semantics:** The `record` expression is used to create new record values. The $exp_i$ are sequentially evaluated and the results are combined in a record value allocated in $Rexp$, or `@=` if omitted. The result is the newly allocated record value.

**Type Information:** The result has type (`recordof` (($n_1$ $Texp_1$) ...($n_m$ $Texp_m$)) $Rexp$) if the type of $exp_i$ is $Texp_i$.

**Effect Information:** The effect of the `record` expression is the `maxeff` of (`alloc` $Rexp$) and the effects of evaluating the $exp_i$.

**Example:**

```
;; Just a standard person record.
;;
(let ((joe (record ((name "Joe L.User")
                    (address "Hacker's Heaven, MIT")
                    (phone-number "256-1000")) @Persons)))
    joe)
```

(`select` *exp* $n_k$)

**Semantics:** The `select` expression accesses a single field of a record value. The first argument must be an expression *exp* of type (`recordof` (($n_1$ $Texp_1$) ...($n_m$ $Texp_m$)) *Rexp*). The field name $n_k$ must be one of the $\{n_j\}$. The result of the `select` expression is the contents of the $n_k$ field of the record value returned by the evaluation of *exp*.

**Type Information:** The type of this expression is $Texp_k$.

**Effect Information:** The effect of this expression is the `maxeff` of the `read` effect on *Rexp* and the effect of evaluating *exp*.

**Example:**

```
;; Taking the address of Joe L. User.
;;
(select joe address)
```

`(record-set!` $exp_1$ $n$ $exp_2$`)`

**Semantics:** The `record-set!` expression mutates a single field of a record value. The first argument must be an expression $exp_1$ of type (`recordof` $((n_1\ Texp_1) \ldots (n_m\ Texp_m))\ Rexp$) where the region $Rexp$ is mutable. The field name $n$ must be one $n_i$ of the $\{n_j\}$. The $exp_1$ is evaluated first and then $exp_2$ is evaluated to yield a value $v$. The field $n_i$ of the record $exp_1$ is then accordingly mutated to the new value $v$. The value returned by the `record-set!` expression is `#u`.

**Type Information:** The type of $exp_2$ must be a subtype of $Texp_i$. The type of the `record-set!` expression is `unit`.

**Effect Information:** The overall effect of the `record-set!` expression is the `maxeff` of the `write` effect on the region $Rexp$ associated with $exp_1$ and the effect of evaluating $exp_1$ and $exp_2$.

***Example:***

```
;; Changing Joe's address.
;;
(record-set! joe address "Big Blue, Yorktown")
```

## 3.18 Oneofs

The notion of tagged variant, or discriminated union, is provided in *FX* by the `oneof` type constructor. There are three special forms for manipulating oneof values: `one`, `tagcase` and `one-set!`.

(**oneof** (($n_1$ $Texp_1$) ($n_2$ $Texp_2$)...($n_m$ $Texp_m$)) $Rexp$)

**Semantics:** This is the type of a oneof value defined in the region $Rexp$ and whose possible tags (identifiers) with associated contents types are ($n_1$ $Texp_1$), ($n_2$ $Texp_2$), ...($n_m$ $Texp_m$).

The tags appearing in the **oneof** type must be distinct. The order of the tags appearing in **oneof** type is irrelevant.

A oneof type (**oneof** (($n_1$ $Texp_{11}$) ($n_2$ $Texp_{12}$) ...($n_m$ $Texp_{1m}$)) $Rexp_1$) is a subtype of (**oneof** (($p_1$ $Texp_{21}$) ($p_2$ $Texp_{22}$) ...($p_q$ $Texp_{2q}$)) $Rexp_2$) iff $m \leq q$, for each field name $n_i$ there is a $j$ such that $n_i = p_j$ and either

1. $Rexp_1$ is a subregion of $Rexp_2$
2. for each field name $n_i$ and corresponding $p_j$, $Texp_{1i}$ is interconvertible to $Texp_{2j}$

or

1. $Rexp_1 = Rexp_2 =$ **@=**
2. for each field name $n_i$ and corresponding $p_j$, $Texp_{1i}$ is a subtype of $Texp_{2j}$

There is no literals for oneof values.

**Kind Information:** The kind of a **oneof** expression is type.

**Example:**

```
;; The type of a basket.
;;
(oneof ((oranges int)
        (apples int))
       @Market)
```

(one *Texp n exp*)

**Semantics:** The one expression is used to create new oneof values. The tag *n* must be one of the tags appearing in *Texp*, which must be a oneof type. A new oneof value of type *Texp* is allocated. The tag of the new value is *n* and its contents is the result of the evaluation of *exp*, the type of which must be a subtype of the type corresponding to *n* in *Texp*. The result of the one expression is the newly allocated oneof value.

**Type Information:** The type of the one expression is *Texp*.

**Effect Information:** The effect of the one expression is the `maxeff` of (`alloc` *Rexp*) and the effect of evaluating *exp*, where *Rexp* is the region parameter of the oneof type *Texp*.

**Example:**

```
;; Creating a oneof representing 3 apples.
;;
(let ((basket (one (oneof ((apples int) (oranges int))
                          @Market)
                   apples 3)))
     basket)
```

```
(tagcase var
         (n_1 exp_11 exp_12 ...)
         (n_2 exp_21 exp_22 ...) ...
         [(else exp_m1 exp_m2 ...)])
                    or
(tagcase (var exp [Rexp_var])
         (n_1 exp_11 exp_12 ...)
         (n_2 exp_21 exp_22 ...) ...
         [(else exp_m1 exp_m2 ...)])
```

All $n_i$ must be distinct.

**Semantics:** The **tagcase** expression selects one of the clauses ($n_i$ $exp_{i1}$ $exp_{i2}$ ...) according to the tag of a oneof value. Two different forms are allowed, either with *var* or (*var exp* [$Rexp_{var}$]). In the first case, the *var* is evaluated and must have a **oneof** type. In the second case, *exp* is evaluated and bound to *var*, which is allocated in $Rexp_{var}$ (or **@=** if omitted); *exp* must have a **oneof** type.

If there is a clause which has the same tag $n_i$ as $v$, the value of *var*, then (begin $exp_{i1}$ $exp_{i2}$ ...) is evaluated in an environment in which *var* is bound to the contents of $v$. Otherwise, (**begin** $exp_{m1}$ $exp_{m2}$ ...) is evaluated.

**Type Information:** Let the **oneof** type of *var* be of the form (**oneof** (($m_1$ $Texp_1$) ($m_2$ $Texp_2$)...) *Rexp*). Then, each tag $n_i$ must be one of the $m_j$. Each $m_j$ must appear exactly once, unless an **else** clause is given, in which case some of the $m_j$ may be omitted. Within the $i$'th clause, the type of *var* is $Texp_j$ if $n_i$ is $m_j$. Within an **else** clause, the type of *var* is unchanged if *Rexp* is mutable, otherwise its **oneof** type is restricted to those fields that don't appear as tags of clauses.

The type of a **tagcase** expression is the maximum of the types of the last expressions in each clause.

**Effect Information:** The effect of the **tagcase** expression is the **maxeff** of (read *Rexp*), (alloc $Rexp_{var}$) and the effects of evaluating $exp_{ij}$ and, if present, *exp*.

**Example:**

```
;; Evaluating the quality of a basket.
;;
(tagcase basket
         (apples "Great !")
         (oranges "Well ..."))
```

**(one-set!** $exp_1$ $n$ $exp_2$**)**

**Semantics:** The one-set! expression mutates the tag and contents of a oneof value, which must be located in a mutable region. $exp_1$ and $exp_2$ are evaluated successively to yield values $v_1$ and $v_2$. The tag and contents of $v_1$ are mutated to $n$ and $v_2$ respectively. The result of the one-set! expression is #u.

**Type Information:** $exp_1$ must have type (oneof (($n_1$ $Texp_1$) ($n_2$ $Texp_2$) ...($n_m$ $Texp_m$)) $Rexp$) where $Rexp$ is a mutable region. The tag $n$ must be the same as some $n_i$. The type of $exp_2$ must be a subtype of $Texp_i$. The type of the one-set! expression is unit.

**Effect Information:** The effect of the one-set! expression is the maxeff of (write $Rexp$) and the effect of evaluating $exp_1$ and $exp_2$.

**Example:**

```
;; Changing the basket to oranges.
;;
(one-set! basket oranges 4)
```

# Chapter 4

# FX Syntactic Sugar

This chapter describes a set of special forms, called *syntactic sugars*, that are not strictly necessary for writing *FX* programs but that represent common idioms of programming. These constructs can be translated, or de-sugared, into other *FX* constructs by simple syntactic transformations. They are provided in the *FX* language as a convenience for the programmer.

The following descriptions are arranged in alphabetical order for easy reference: **and**, `cond`, `dlet`, `dlet*`, **do**, `let`, `let*`, `plet*` and **or**.

$$(\textbf{and}\ \mathit{exp}_1 \ldots \mathit{exp}_n)$$

**Semantics:** This form performs the "and" evaluation of boolean expressions using a "short-circuit" technique.

Every expression $\mathit{exp}_i$ is successively evaluated. As soon as `#f` is returned by one $\mathit{exp}_i$, the evaluation of the (possibly) remaining expressions is abandoned and `#f` is returned. If all the $\mathit{exp}_i$ return `#t`, then `#t` is the value of the and expression. By convention, (and) evaluates to `#t`.

**Type Information:** Each $\mathit{exp}_i$ must be of type `bool`, which is the type of the whole expression.

**Effect Information:** The effect of the **and** construct is the `maxeff` of the effects of evaluating $\mathit{exp}_i$.

**Sugar Information:** The above **and** expression is equivalent to:

$$(\texttt{if}\ \mathit{exp}_1\ (\texttt{if} \ldots (\texttt{if}\ \mathit{exp}_n\ \texttt{\#t}\ \texttt{\#f}) \ldots)\ \texttt{\#f})$$

**Example:**

```
;; A test for valid vector indices.
;;
(and (>= index 0)
     (< index (vector-length v))
     (= (vector-ref v index) 0))
```

$$
\begin{array}{l}
(\texttt{cond}\ (exp_{test1}\ exp_{11}\ \ldots exp_{1n}) \\
\qquad\quad \ldots \\
\qquad\quad (exp_{testk}\ exp_{k1}\ \ldots exp_{km}) \\
\qquad\quad (\texttt{else}\ exp_{(k+1)1}\ \ldots exp_{(k+1)p}))
\end{array}
$$

**Semantics:** The cond special form is a multiple-way test expression. The expressions $exp_{testj}$ are successively evaluated and as soon as one returns `#t` (or the `else` clause is reached), the associated expressions are evaluated as if they were inclosed in a `begin` special form.

**Type Information:** The expressions $exp_{testi}$ must be of type `bool`. The type of a cond expression is the maximum of the types of the last expressions in each branch (namely $exp_{1n}\ \ldots$).

**Effect Information:** The effect of the cond construct is the `maxeff` of the effects of all the $exp_{testi}$ and $exp_{ij}$.

**Sugar Information:** The above cond expression is equivalent to:

$$
\begin{array}{l}
(\texttt{if}\ exp_{test1} \\
\quad (\texttt{begin}\ exp_{11}\ \ldots exp_{1n}) \\
\quad (\texttt{if}\ \ldots \\
\qquad \ldots \\
\qquad (\texttt{if}\ exp_{testk} \\
\qquad\quad (\texttt{begin}\ exp_{k1}\ \ldots exp_{km}) \\
\qquad\quad (\texttt{begin}\ exp_{(k+1)1}\ \ldots exp_{(k+1)p}))\cdots))
\end{array}
$$

**Example:**

```
;; The addition function.
;;
(letrec ((add (lambda ((n int) (m int))
                (the pure int
                     (cond ((< n 0) 0)
                           ((= n 0) m)
                           (else (add (- n 1)
                                      (+ m 1))))))))
        (add 1 2))
```

(**dlet** (($d_1$ $Dexp_1$)...) $Dexp_{body}$)

The $d_i$ must all be distinct.

**Semantics:** **dlet** provides a means of introducing synonyms for type, effect, and region expressions. The value of a **dlet** description expression is a description expression, namely $Dexp_{body}$ with the $d_i$ replaced by the corresponding $Dexp_i$.

**Kind Information:** The kind of the **dlet** description expression is the kind of $Dexp_{body}$ with each $d_i$ replaced by $Dexp_i$.

**Sugar Information:** The **dlet** description expression is equivalent to:

((**dlambda** (($d_1$ $Kexp_1$)...) $Dexp_{body}$) $Dexp_1$...)

assuming that the kind of $Dexp_i$ is $Kexp_i$.

**Example:**

```
;; The type of a subroutine which takes two binary operations
;; on bools, two booleans and returns whichever one returns
;; #t (say).
;;
(dlet ((subr-type (subr pure (bool bool) bool)))
      (subr pure (subr-type subr-type bool bool) subr-type))
```

$$\texttt{(dlet* ((}d_1\ \mathit{Dexp}_1\texttt{)}\ldots\texttt{)}\ \mathit{Dexp}_{body}\texttt{)}$$

**Semantics:** **dlet*** provides a means of introducing synonyms for type, effect, and region expressions.

The value of a **dlet*** description expression is the value of its body. Whenever $d_i$ is encountered in the **dlet*** body, it is replaced by $\mathit{Dexp}_i$. A reference to a $d_i$ in $\mathit{Dexp}_j$ is taken to refer to, either a previous binding of $d_i$ in the current **dlet*** (if such binding exists), or a binding for $d_i$ in the surrounding (outer) scope.

**Kind Information:** The kind of the **dlet*** description expression is the kind of $\mathit{Dexp}_{body}$ with each $d_i$ replaced by $\mathit{Dexp}_i$.

**Sugar Information:** The **dlet*** description expression is equivalent to:

$$\texttt{(dlet ((}d_1\ \mathit{Dexp}_1\texttt{)) (dlet* (}\ldots\texttt{)}\ \mathit{Dexp}_{body}\texttt{))}$$

**Example:**

```
;; The type of a subroutine which takes two binary operations
;; on bools, two booleans and returns whichever one returns
;; #t (say).
;;
(dlet* ((t bool)
        (subr-type (subr pure (t t) t)))
       (subr pure (subr-type subr-type t t) subr-type))
```

$$\begin{array}{l}(\texttt{do}\ ((var_1\ exp_{init1}\ [exp_{step1}\ [Rexp_1]])\ldots)\\ \quad (exp_{test}\ exp_{rtn1}\ exp_{rtn2}\ldots exp_{rtnn})\\ \quad exp_{body1}\ldots)\end{array}$$

The $var_i$ must all be distinct.

Omitting $exp_{stepi}$ is the same as writing $var_i$ there (*i.e.*, $(var_i\ exp_{initi})$ is the same as $(var_i\ exp_{initi}\ var_i)$).

**Semantics:** do expressions are used for performing iterations. They define a set of iteration variables, which are new variables allocated in $Rexp_i$ (or @= if omitted), with initial values for those variables and (optionally) expressions for updating them.

There are three parts to the evaluation of a do expression: the initialization, the iteration, and the return.

During the initialization, the $exp_{initi}$ are evaluated from left to right and the resulting values are bound to the corresponding $var_i$. These $var_i$ are not available for use in the $exp_{initi}$.

At the beginning of every iteration, $exp_{test}$ is evaluated. If the test returns `#f`, then the $exp_{bodyi}$ are evaluated in order. Then, the $exp_{stepi}$ are evaluated from left to right and the results are bound as new values for the $var_i$. Iteration then starts over again.

If $exp_{test}$ returns `#t` at the beginning of an iteration, then the result of the evaluation of (`begin` $exp_{rtn1}$ $exp_{rtn2}$ ...) is returned.

**Type Information:** The expression $exp_{test}$ must be of type `bool`. Each $var_i$ has the same type as $exp_{initi}$. The type of each $exp_{stepi}$ must be a subtype of the type of $var_i$. The type $Texp_{do}$ of the do expression is the type of the last return expression, $exp_{rtnn}$.

**Effect Information:** The effect $Eexp_{do}$ of a do expression is computed by performing effect masking on the `maxeff` of the effects of the $exp_{initi}$, the $exp_{stepi}$, $exp_{test}$, the $exp_{rtni}$, the $exp_{bodyi}$, and (`alloc` $Rexp_i$).

**Sugar Information:** The above do expression is equivalent to:

```
(letrec ((do-temp
           (lambda ((var₁ Texp₁ Rexp₁)...)
                   (the Eexp_do Texp_do
                        (if exp_test
                            (begin exp_rtn1
                                   exp_rtn2 ...
                                   exp_rtnn)
                            (begin exp_body1 ...
                                   (do-temp exp_step1 ...)))))))
        (do-temp exp_init1 ...))
```

where *$Texp_i$* is the type of *$exp_{initi}$* and do-temp is a new identifier.

**Example:**

```
;; Loop until foo? is verified and then return the toggle.
;;
(do ((toggle #t (not toggle)))
    ((foo? toggle) toggle)
    )
```

(let (($var_1$ $exp_1$ [$Rexp_1$])...) $exp_{1b}$ $exp_{2b}$...)

The body of the let expression, $exp_{1b}$ $exp_{2b}$..., is treated as though (begin $exp_{1b}$ $exp_{2b}$...) is written.

The $var_i$ must all be distinct.

**Semantics:** let provides a way for creating synonyms, or shorthand names, for complicated, lengthy, or computationally expensive expressions.

The body of the let expression, the expressions $exp_{jb}$, is evaluated with each of the variables $var_i$, allocated in $Rexp_i$ (or @= if omitted), denoting the value resulting from the evaluation of its corresponding $exp_i$. A reference to one of the $var_i$ within these expressions is interpreted as a reference to a binding for that $var_i$ in the surrounding (outer) scope of let (see the description of letrec for a discussion of recursion).

The value of the let expression is the value of its body evaluated in this way.

**Type Information:** The type of the let expression is the type of its body.

**Effect Information:** The effect of a let expression is the maxeff of the effects of the $exp_i$, the $exp_{ib}$, and (alloc $Rexp_i$).

**Sugar Information:** A let expression is equivalent to an application of a lambda expression. The above let expression is equivalent to:

((lambda (($var_1$ $Texp_1$ [$Rexp_1$])...) $exp_{1b}$ $exp_{2b}$...) $exp_1$...)

*assuming that the types of $exp_i$ is $Texp_i$.*

**Example:**

```
;; Avoid a double (expensive) computation.
;;
(let ((y (expensive-computation)))
     (foo y y))
```

(let* (($var_1$ $exp_1$ [$Rexp_1$])...) $exp_{1b}$ $exp_{2b}$...)

The body of the let* expression, $exp_{1b}$ $exp_{2b}$..., is treated as though (begin $exp_{1b}$ $exp_{2b}$...) is written.

**Semantics:** let* provides a way for creating synonyms, or shorthand names, for complicated, lengthy, or computationally expensive expressions.

The body of the let* expression, the expressions $exp_{jb}$, is evaluated with each of the $var_i$, allocated in $Rexp_i$ (or @= if omitted), denoting the value resulting from the evaluation of its corresponding $exp_i$. A reference to one of the $var_i$ within these expressions is interpreted as a reference to, either a previous binding of $var_i$ in the current let* (if such binding exists), or a binding for that $var_i$ in the surrounding (outer) scope of let* (see the description of letrec for a discussion of recursion.)

The value of the let* expression is the value of its body evaluated in this way.

**Type Information:** The type of the let* expression is the type of its body.

**Effect Information:** The effect of a let* expression is computed by performing effect masking on the maxeff of the effects of the $exp_i$, the $exp_{ib}$, and (alloc $Rexp_i$).

**Sugar Information:** A let* expression is equivalent to a nested list of let. The above let* expression is equivalent to:

(let (($var_1$ $exp_1$ [$Rexp_1$]))
    (let* (...) $exp_{1b}$ $exp_{2b}$...))

**Example:**

```
;; Avoid a double (expensive and voluminous) computation.
;;
(let* ((y (voluminous-expression))
       (y (expensive-computation y)))
      (foo y y))
```

(or $exp_1 \ldots exp_n$)

**Semantics:** This form performs the "or" evaluation of boolean expressions using a "short-circuit" technique.

Every expression $exp_i$ is evaluated in turn proceeding from left to right. As soon as one of the $exp_i$ evaluates to `#t`, the evaluation of the (possibly) remaining expressions is abandoned and `#t` is returned. If all the $exp_i$ return `#f`, then `#f` is returned. By convention, (or) evaluates to `#f`.

**Type Information:** All of the $exp_i$ must be of type `bool`. The type of an or expression is `bool`.

**Effect Information:** The effect of the or construct is the `maxeff` of the effects of evaluating the $exp_i$.

**Sugar Information:** The above or expression is equivalent to:

(`if` $exp_1$ `#t` (`if` … (`if` $exp_n$ `#t` `#f`)…))

**Example:**

```
;; A debug-only test.
;;
(or *debug-phase*
    (to-be-tested-latter))
```

(**plet*** (($d_1$ $Dexp_1$)...) $exp_1$ $exp_2$...)

The body of the **plet*** expression, $exp_1$ $exp_2$..., is treated as though (**begin** $exp_1$ $exp_2$...) is written.

**Semantics:** **plet*** provides a way of making type, effect, region and description function synonyms, or shorthand names, for complicated description expressions.

The value of a **plet*** value expression is the value of its body. Whenever $d_i$ is encountered in the **plet*** body, it is replaced by $Dexp_i$. A reference to a $d_i$ in $Dexp_j$ is taken to refer to, either a previous binding of $d_i$ in the current **plet*** (if such binding exists), or a binding for $d_i$ in the surrounding (outer) scope. (See the description of **pletrec** for a discussion of recursive types.)

**Type Information:** The type of the **plet*** expression is the type of its body with $Dexp_i$ substituted for $d_i$.

**Effect Information:** The effect of the **plet*** expression is the effect of its body with $Dexp_i$ substituted for $d_i$.

**Sugar Information:** The **plet*** value expression is equivalent to:

(**plet** (($d_1$ $Dexp_1$)) (**plet*** (...) $exp_1$ $exp_2$...))

**Example:**

```
;; The identity function on a complicated type.
;;
(plet* ((t (subr pure (bool) bool))
        (t (ref t @!)))
       (lambda ((a-ref-to-subr t)) a-ref-to-subr))
```

# Chapter 5

# The FX Environment

We describe in this chapter the *FX* programming environment, which includes input and output operations, an error-signalling facility, an interpreter-oriented top-level, and a simple way to structure large *FX* programs. These primitive facilities will be supported by every *FX* implementation.

## 5.1 I/O Facilities

The I/O functions deal with a new value in the *FX* system, namely the file system.

### Definitions

This subsection introduces two new types: `input-port` and `output-port`, both of kind `type`. *FX* uses *ports* as an abstraction for files, where values of type `input-port` are used for read operations and `output-port` for write operations.

I/O operations have effects on the `@IO` region. The region `@IO` is used to describe the state of the file system and the input and output file pointers.

### Literals

There are no literals of type `input-port` or `output-port`.

## Operations

*FX* provides a set of I/O subroutines that are compatible with the Scheme I/O primitives. Since the *FX* input and output subroutines are duals of one another, we only define the types of input subroutines. In order to compute the types of output subroutines (the names of which are given in braces), replace input by output in the definitions.

```
call-with-input-file {call-with-output-file}:
  (poly ((r region))
    (poly ((t type) (e effect))
          (subr (maxeff e (alloc @IO) (read r))
                ((string r) (subr e (input-port) t))
                t)))
```

```
current-input-port {current-output-port} :
   (subr (maxeff (write @IO) (read @IO)) () input-port)
```

Every running *FX* program inherits from the environment an initial value for (current-input-port) and (current-output-port), initially bound to the keyboard and the console.

```
with-input-from-file {with-output-to-file} :
  (poly ((r region))
    (poly ((t type) (e effect))
          (subr (maxeff (write @IO) (read @IO)
                        e (read r) (alloc @IO))
                ((string r) (subr e () t))
                t)))
```

```
open-input-file {open-output-file} :
  (poly ((r region))
        (subr (maxeff (alloc @IO) (read r)
                      (write @IO) (read @IO))
              ((string r)) input-port))
```

```
close-input-port {close-output-port} :
  (subr (maxeff (read @IO) (write @IO)) (input-port) unit)
```

```
char-ready? :
  (vsubr (maxeff (read @IO) (write @IO)) input-port bool)
```

Read functions are provided for each basic type. A dynamic error will be signalled if the type of the value to be read is not correct. These read functions use the current input-port to perform their I/O operation.

```
read-bool :
  (subr (maxeff (write @IO) (read @IO)) () bool)
read-char :
  (subr (maxeff (write @IO) (read @IO)) () char)
read-int :
  (subr (maxeff (write @IO) (read @IO)) () int)
read-float :
  (subr (maxeff (write @IO) (read @IO)) () float)
read-string :
  (subr (maxeff (write @IO) (read @IO)) () (string @=))
read-symbol :
  (subr (maxeff (write @IO) (read @IO)) () symbol)
```

It is a dynamic error to perform a read operation if the end of file of the current input port is reached. The presence of the end of file for the current input port can be tested by the **eof?** function:

```
eof?  :  (subr (maxeff (write @IO) (read @IO)) () bool)
```

*FX* also provides a set of type specific write subroutines which use the current output-port.

```
write-bool :
  (subr (maxeff (write @IO) (read @IO)) (bool) unit)
write-char :
  (subr (maxeff (write @IO) (read @IO)) (char) unit)
write-int :
  (subr (maxeff (write @IO) (read @IO)) (int) unit)
write-float :
  (subr (maxeff (write @IO) (read @IO)) (float) unit)
write-string :
  (poly ((r region))
        (subr (maxeff (write @IO) (read @IO) (read r))
              ((string r))
              unit))
write-symbol :
  (subr (maxeff (write @IO) (read @IO)) (symbol) unit)
```

In addition to these subroutines for specific data types, *FX* provides general symbolic expression input and output via the **sexp** type:

```
(dletrec ((sexp (oneof ((s-unit unit)
                        (s-bool bool)
                        (s-int int)
                        (s-float float)
                        (s-char char)
                        (s-symbol symbol)
                        (s-string (string @=))
                        (s-vectorof (vectorof sexp @=))
                        (s-null null)
                        (s-pairof (pairof sexp sexp @=)))
                       @=)))
         sexp)
```

Two subroutines are provided to read a write **sexp** values:

```
read-sexp  :  (subr (maxeff (read @IO) (write @IO)) () sexp)
write-sexp :  (subr (maxeff (read @IO) (write @IO)) (sexp) unit)
```

Every **sexp** value has a literal value which when written by **write-sexp** can be read back in by **read-sexp**. The concrete syntax used for **sexp** literals is the one used by *FX* for **unit**, **bool**, **int** (in base 10), **float**, **char**, (**string @=**) and **null** values. **symbol** literals are identifiers with the same name as the symbol. **vectorof** literals are be enclosed between the delimiters **#(** and **)**. **pairof** literals are enclosed between the delimiters **(** and **)**.

## 5.2 Signalling Errors

The error subroutine displays an optional message and signals a dynamic error. A call to error does not return and is handled in an unspecified manner.

```
error :  (poly ((r region))
               (subr (read r) ((string r)) void))
```

Note that there is no way to continue a computation once error is called.

## 5.3 Top-Level

The *FX* top-level is a *read-check-eval-print* loop. As each expression is input by the user, it is processed by each of the four stages of this loop:

- The *reader* reads the expression and checks its syntax.
- The *type and effect system* checks the type and the effect of the expression. If a static error is detected the user is informed and control is returned to the reader which waits for another expression.
- The *evaluator* evaluates the expression and computes its result value.
- The *printer* outputs the result value (in an unspecified format), and calls the reader which waits for another expression.

The current state of the *FX* interpreter is defined by the *definition environment*, which is a list of description definitions (bindings of description variables to description values) and ordinary definitions (bindings of ordinary variables to ordinary values).

Additional top-level definitions can be created by the `define` and `pdefine` *top-level* special forms, respectively:

- A `define` special form binds (or rebinds) new values to top-level variables;
- A `pdefine` special form binds new description values to description variables.

The *FX* top-level process is structured as the analysis of a sequence of *definition blocks* and ordinary expressions, which may be interleaved arbitrarily. A definition block is a sequence of `pdefine` and `define` expressions. The *FX* type and effect system stays inactive while a definition block is being entered. As soon as a definition block is complete it is evaluated in order to update the current definition environment. A definition block is considered to be complete either when an ordinary (value) expression is entered, or when the *FX* top-level detects that no pending undefined variables remain.

When a definition block is complete it is evaluated in the following manner. First, every new description binding is added to the list of description definitions. It is a static error to attempt to rebind a description variable if the new description is not convertible to the old one. Second, every new ordinary binding is added to the list of ordinary definitions. It is a static error to attempt to rebind an ordinary variable if the type of the new value is not a subtype of the type of the old one.

If a static error occurs during the evaluation of a definition block, the definition block is abandoned and the definition environment is restored to its state before the present definition block was entered.

When ordinary expressions $exp_1$ ... are entered after zero or more definition blocks, they are evaluated sequentially as if in the body of a `pletrec` expression formed with the list of description definitions present in the updated definition environment, and a `letrec` expression formed with the list of ordinary definitions from the updated definition environment.

The initial definition environment includes all of the standard types and variables defined in this manual.

(**define** *var exp*)
or
(**define** (*var* (*var*$_1$ *Texp*$_1$ [*Rexp*$_1$]) ...) *exp*$_1$ *exp*$_2$...)

The second form is equivalent to

(**define** *var* (**lambda** ((*var*$_1$ *Texp*$_1$ [*Rexp*$_1$]) ...) *exp*$_1$ *exp*$_2$...))

and so, the following description only deals with the first form.
This special form is allowed only at the top-level of *FX*.

**Semantics:** The **define** special form extends the current set of ordinary variable definitions which are visible at the top-level. Specifically *var* is bound in the region **@=** to the value of the expression *exp*. If *var* is already bound, the previous binding is lost and the other ordinary definitions that used the previous binding of *var* now refer to this new version.

If *var* is already bound, the type of *exp* must be a subtype of the type of *var*.

At any given time, the set of defined ordinary variables is logically embedded in an enclosing **letrec** form, so the restrictions imposed by this form must be observed when using the **define** special form.

**Examples:**

```
;; An add2 function with a forward reference.
;;
(define add2 (lambda ((x int))
                  (the pure int
                       (add1 (add1 x)))))

;; And now, the add1 function.
;;
(define (add1 (x int))
        (the pure int (+ x 1)))
```

**(pdefine** ***d Dexp*****)**
**or**
**(pdefine (*****d*** **(*****d*****$_1$** ***Kexp*****$_1$) ...)** ***Dexp*****)**

The second form is equivalent to

**(pdefine** ***d*** **(dlambda ((*****d*****$_1$** ***Kexp*****$_1$) ...)** ***Dexp*****))**

and so, the following description will only deal with the first form.

This special form is allowed only at the top-level of *FX*.

**Semantics:** This special form extends the current set of description variable definitions which are visible at the top-level; specifically, *d* is bound to the description expression *Dexp*.

If *d* is already bound, *Dexp* must be convertible to the old value of *d*.

At any given time, the set of defined description variables is logically embedded in an enclosing `pletrec` form, so the restrictions imposed by this form must be observed when using the `pdefine` special form.

**Example:**

```
;; Define the list-of-int type.
;;
(pdefine list-of-int (listof int @my-lists))
```

## 5.4 Structuring programs

The *FX* user may want to use different *FX* files to develop large *FX* programs. The following special forms provide the ability to deal with this kind of incremental program development: `load` and `compile`.

(load ***string-literal***)

The load special form is allowed only at the top-level of *FX*.

**Semantics:** The load special form enters the contents of the specified file into *FX* as if they had been typed interactively at top-level, except in the way that *FX* behaves if an error is signalled. If an error is signalled during a load operation, the current definition block is abandoned and the definition environment is restored to what it was before the current block; the top-level is then restarted. load can be included in files that are loaded. The format of the file name string is not specified.

**Example:**

```
;; Loading the init.fx file.
;;
(load "psrg:>fx>init.fx")
```

(`compile` ***string-literal***)

The `compile` special form is allowed only at the top-level of *FX*.

**Semantics:** The `compile` special form creates an optimized version of the program in the specified file. This optimized version will be used the next time the specified file is loaded. The `compile` form does not alter the definition environment. The format of the file name string is not specified.

**Example:**

```
;; Compile the init.fx file.
;;
(compile "psrg:>fx>init.fx")
```

# Appendix A

# FX Syntax

This appendix gives the syntax of *FX* as a BNF grammar. For the purposes of this grammar, *id* denotes an identifier, *var* a variable, *integer* a non-empty sequence of digits and *character* a character.

*Kexp* = – Kind expressions

    **region** | **effect** | **type**
    (**dfunc** (*Kexp*...) *Kexp*)

*Dexp* = – Description expressions

    *Rexp* | *Eexp* | *Texp* | *HDesc*

*HDesc* = – Higher order description expressions

    *GDesc*
    (**dlambda** ((*var Kexp*)...) *Dexp*)

*GDesc* = – Generic description expressions

    *var*
    (*HDesc Dexp*...)
    (**dlet** ((*var Dexp*)...) *Dexp*)
    (**dlet*** ((*var Dexp*)...) *Dexp*)
    (**dletrec** ((*var Dexp*)...) *Dexp*)

*Rexp* = – **Region expressions**

```
    GDesc
    @id
    (runion Rexp Rexp ...)
```

*Eexp* = – **Effect expressions**

```
    GDesc
    pure
    (alloc Rexp)
    (read Rexp)
    (write Rexp)
    (maxeff Eexp...)
```

*Texp* = – **Type expressions**

```
    GDesc
    bool | char | float | int
    null | symbol | unit| void
    (subr Eexp (Texp...) Texp)
    (poly ((var Kexp)...) Texp)
    (ref Texp Rexp)
    (string Rexp)
    (pairof Texp Texp Rexp)
    (listof Texp Rexp)
    (vsubr Eexp Texp Texp)
    (promise Eexp Texp)
    (uniqueof Texp)
    (vectorof Texp Rexp)
    (recordof ((var Texp) ...) Rexp)
    (oneof ((var Texp) (var Texp)...) Rexp)
```

*TopLevel* = – **Top Level Value expressions**

```
    (define var exp)
    (define (var (var Texp [Rexp])...) exp exp...)
    (pdefine var Dexp)
    (pdefine (var (var Kexp)...) Dexp)
```

```
        (load "character...")
        (compile "character...")


exp =                           – Value expressions
        var
        Literal
        (exp exp...)
        (and exp...)
        (begin exp exp...)
        (cond (exp exp exp...)...(else exp exp...))
        (delay exp)
        (do ((var exp [exp [Rexp]])...) (exp exp exp...) exp...)
        (if exp exp exp)
        (lambda ((var Texp [Rexp])...) exp exp...)
        (let ((var exp [Rexp])...) exp exp...)
        (letrec ((var exp [Rexp])...) exp exp...)
        (let* ((var exp [Rexp])...) exp exp...)
        (one Texp var exp)
        (one-set! exp var exp)
        (or exp...)
        (plambda ((var Kexp)...) exp)
        (plet ((var Dexp)...) exp exp...)
        (pletrec ((var Dexp)...) exp exp...)
        (plet* ((var Dexp)...) exp exp...)
        (proj exp Dexp...)
        (record ((var exp)...) [Rexp])
        (record-set! exp var exp)
        (select exp var)
        (set! var exp)
        (tagcase var (var exp exp...) (var exp exp...)...
                    [(else exp exp...)])
        (tagcase (var exp [Rexp]) (var exp exp ...) (var exp exp...)...
                                   [(else exp exp ...)])
        (the [Eexp] Texp exp)
        (vlambda (var Texp [Rexp]) exp exp...)
```

*Literal* = – Literal expressions

- **#u**
- **#t** | **#f**
- [**#***Base*][*Sign*]*integer*
- [*Sign*]*integer* . *integer* [e[*Sign*]*integer*]
- **#\***Char*
- **"***character...***"**
- '*id*
- (quote *id*)
- **()**

*Base* = **b** | **o** | **d** | **x**

*Sign* = **+** | **-**

*Char* = *character* | **backspace** | **newline** | **page** | **space** | **tab**

# Appendix B

# FX Semantics

This appendix describes the constructs and concepts that form the basis of the *FX* language. In particular, we state the following claims: type soundness, static typing, location invariance, untyped semantics, typeless implementation, and effect soundness. The proofs are omitted.

To keep the presentation simple, many features not essential to the theory behind *FX* have been omitted from this appendix. These include immutable regions, description functions, multiple function arguments, implicit `begin` expressions (*e.g.* inside `lambda`), recursion, built-in types such as integers and strings, data structuring types such as records and oneofs, I/O functions and assignable variables.

## B.1 Grammar

The grammar of the language described in this appendix is given below, starting with kinds and proceeding to descriptions and ordinary expressions. For the most part, this grammar generates the same language as that described in the previous appendix. We reproduce an entire grammar here so that we may simultaneously introduce some new notation and the simplifications alluded to above.

The meta-variable for each syntactic class is shown in parentheses. Meta-variables in this appendix differ from those in the rest of the manual so that the formal rules presented here have a more compact representation.

*Kinds* serve as the "types" of descriptions. There are three kinds, corresponding to the three kinds of descriptions — region, effect, and type descriptions.

| | |
|---|---|
| *Kind* = | – kinds ($\kappa$) |
| `region` | – kind of regions |
| `effect` | – kind of effects |
| `type` | – kind of types |

*Descriptions* serve to describe the types and effects of ordinary expressions. We discuss the three kinds of descriptions — region, effect, and type descriptions — in turn, but first we define the description constants and variables.

| | |
|---|---|
| *Rconst* $= \{r_1, r_2, \ldots\}$ | – region constants ($r$) |
| *Tconst* $=$ {`unit`, `bool`} | – type constants ($t$) |
| *Econst* $=$ {`pure`} | – effect constants |
| *Dvar* $= \{d_1, d_2, \ldots\}$ | – description variables ($d$) |

*Region descriptions* correspond to countably infinite sets of locations, which we call *regions*. The `runion` of one or more region descriptions corresponds to the *union* of the corresponding sets of locations. The precise meaning of "locations" is given on page 134.

| | |
|---|---|
| *Region* = | – region descriptions ($\rho$) |
| *Rconst* | – region constant |
| *Dvar* | – description variable |
| (`runion` *Region*$^+$) | – union of one or more regions |

*Effect descriptions* correspond to the side-effects of ordinary expressions, as expressed in terms of the allocating, reading, and writing of locations in certain regions. The `maxeff` of zero or more effect descriptions corresponds to the *combination* of the corresponding effects.

| | |
|---|---|
| *Effect* = | – effect descriptions ($\epsilon$) |

| | |
|---|---|
| *Econst* | – effect constant |
| *Dvar* | – description variable |
| (`alloc` *Region*) | – allocate in a region |
| (`read` *Region*) | – read from a given region |
| (`write` *Region*) | – write to a given region |
| (`maxeff` *Effect**) | – combine zero or more effects |

*Type descriptions* correspond to sets of values, which we call *types*. For example, the type constant `unit` corresponds to the set {`#u`} and the type constant `bool` corresponds to the set {`#t`, `#f`}. The precise meaning of "values" is given on page 121.

| | | |
|---|---|---|
| *Type* = | | – type descriptions ($\tau$) |
| | *Tconst* | – type constant |
| | *Dvar* | – description variable |
| | (`subr` *Effect* (*Type*) *Type*) | – ordinary subroutines |
| | (`poly` (*Dvar Kind*) *Type*) | – polymorphic subroutines |
| | (`ref` *Type Region*) | – locations |

The description (`subr` $\epsilon$ ($\tau_1$) $\tau_2$) is a generalization of the type $\tau_1 \rightarrow \tau_2$ in the typed lambda-calculus. It corresponds to the set of ordinary subroutines that, when applied to a value in $\tau_1$, cause a side-effect of at most $\epsilon$ and either diverge or return a value whose type is a subtype of $\tau_2$. The term "subtype" is defined on page 122.

The description (`poly` ($d$ $\kappa$) $\tau$) is a generalization of the type $\forall t.\tau$ in the second-order typed lambda-calculus. It corresponds to the set of polymorphic expressions that, when projected onto a description $\delta$ of kind $\kappa$, either diverge or return a value whose type is at most $\tau[\delta/d]$, without causing any side-effects. The post-fix $[\delta/d]$ denotes substitution of $\delta$ for $d$, where bound variables are renamed as needed to avoid capture.

The description (`ref` $\tau$ $\rho$) is a generalization of the type *ref* $\tau$ in programming languages such as ML. It corresponds to the set of locations in the region $\rho$ that are intended for values whose type is a subtype of $\tau$.

The grammar for descriptions in general is obtained by combining the region, effect, and type descriptions.

| | | |
|---|---|---|
| *Desc* = | | – descriptions ($\delta$) |
| | *Region* | – region descriptions |
| | *Effect* | – effect descriptions |
| | *Type* | – type descriptions |

*Ordinary expressions* serve to express programs and the resulting values. We discuss the ordinary expressions below, but first we define the ordinary constants and variables.

| | | |
|---|---|---|
| *Unit* = | {**#u**} | – the unit type |
| *Bool* = | {**#t**, **#f**} | – Booleans ($b$) |
| *Var* = | $\{x_1, x_2, \ldots\}$ | – ordinary variables ($x$) |

The constants of the language are **#u**, **#t**, and **#f**.

| | | |
|---|---|---|
| *Const* = | | – ordinary constants ($c$) |
| | *Unit* | – the unit type |
| | *Bool* | – the Booleans |

The grammar for ordinary expressions in general is given below. There are three general classes of ordinary expressions: expressions that derive from the second-order typed lambda-calculus; expressions that deal with evaluation order; and expressions that deal with side-effects. The first class consists of constants and variables, ordinary abstraction and application, and polymorphic abstraction and projection. The second class consists of expressions for conditional and sequential evaluation. The third class consists of expressions for asserting type and effect and for the allocating, reading, and writing of locations.

| | | |
|---|---|---|
| *exp* = | | – ordinary expressions ($e$) |
| | *Const* | – ordinary constants |

| | |
|---|---|
| *Var* | – ordinary variables |
| `(lambda (`*Var Type*`)` ***exp***`)` | – ordinary abstraction |
| `(`***exp exp***`)` | – ordinary application |
| `(plambda (`*Dvar Kind*`)` ***exp***`)` | – polymorphic abstraction |
| `(proj` ***exp*** *Desc*`)` | – polymorphic application |
| `(if` ***exp exp exp***`)` | – conditional evaluation |
| `(begin` ***exp***$^+$`)` | – sequential evaluation |
| `(the` *Effect Type* ***exp***`)` | – effect/type assertion |
| `(new` *Type Region* ***exp***`)` | – allocating a location |
| `(get` ***exp***`)` | – reading a location |
| `(set` ***exp exp***`)` | – writing a location |

Notice that `new`, `get`, and `set` are special forms here while they were polymorphic subroutines in the main body of the manual. This allows explanation of semantics involving `new`, `get`, and `set` without the complexity of projecting them.

Certain ordinary expressions, such as applications, represent *computations*; other ordinary expressions, such as constants, do not exhibit any computational behavior, and represent *values*.

**Definition.** An ordinary expression is a *value* iff it is a constant, a `lambda` expression, or a `plambda` expression. We use *Val* to denote the set of values and $v$ to denote individual values. In other words,

| | | |
|---|---|---|
| *Val* = | | – values ($v$) |
| | *Const* | – ordinary constant |
| | `(lambda (`*Var Type*`)` ***exp***`)` | – ordinary abstraction |
| | `(plambda (`*Dvar Kind*`)` ***exp***`)` | – polymorphic abstraction |

## B.2 Free Variables and Constants

Free and bound variables are defined as in the second-order typed lambda-calculus; in particular, `lambda` binds ordinary variables, and `plambda` and `poly` bind description variables. This is formalized in the following definitions.

**Definition.** The free ordinary variables of an ordinary *expression* are given by the function $FV : exp \rightarrow \mathrm{PowerSet}(Var)$.

**Definition.** The free description variables of a description expression are given by the function $FDV : Desc \rightarrow \mathrm{PowerSet}(\mathbf{Dvar})$.

**Definition.** The free description variables of an ordinary expression are given by the function $FDV : exp \rightarrow \mathrm{PowerSet}(\mathbf{Dvar})$.

We adopt the usual notions of alpha-renaming and beta-substitution for bound variables. We use the notation $\delta_1[\delta_2/d]$, $e[\delta/d]$ and $e_1[e_2/x]$ to indicate beta-substitution, where bound variables are renamed as needed to avoid capture. We adopt the usual definition of closed descriptions and ordinary expressions:

**Definition.** A *description* $\delta$ is *closed* iff it has no free description variables, *i.e.* iff $FDV(\delta) = \phi$.

**Definition.** An ordinary expression $e$ is *closed* iff it has no free ordinary variables and no free description variables, *i.e.* iff $FV(e) = \phi \wedge FDV(e) = \phi$.

It is convenient to define the free region constants of a description, $FRC(\delta)$, and of an ordinary expression, $FRC(e)$. Since region constants cannot be bound, the definition of $FRC$ and $FRC$ is trivial: all the region constants that occur in a description or ordinary expression are free.

**Definition.** The free region constants of a description are given by the function $FRC : Desc \rightarrow \mathrm{PowerSet}(\mathbf{Rconst})$.

**Definition.** The free region constants of an ordinary expression are given by the function $FRC : exp \rightarrow \mathrm{PowerSet}(\mathbf{Rconst})$.

## B.3 Description Conversion and Inclusion

In this section, we define the *conversion* and *inclusion* relations on descriptions. The conversion relation ($\simeq$) is an equivalence relation that partitions *Desc* into sets of descriptions that correspond to the same underlying sets of locations, effects, or values. Two descriptions can be convertible only if they have the same kind.

The inclusion relation ($\sqsubseteq$) is a partial order that relates pairs of descriptions that correspond to sets such that one is a subset of the other. Thus, one region is a subregion of another if the set of locations corresponding to the former is a subset of the set of locations corresponding to the latter. This rule applies to effects and types as well. Two descriptions can be related only if they have the same kind; it follows that the description inclusion relation is simply a combination of the independent subregion, subeffect, and subtype relations. These inclusion relations are defined below. The conversion relation then follows from the definition of inclusion:

$$\delta_1 \simeq \delta_2 \quad \Leftrightarrow \quad (\delta_1 \sqsubseteq \delta_2) \wedge (\delta_1 \sqsupseteq \delta_2)$$

The conversion and inclusion relations are determined completely by the correspondence between descriptions and sets of locations, effects or values respectively.

## Region Descriptions

As previously mentioned, a region description corresponds to a countably infinite set of locations. We define the inclusion relation on regions to correspond to set inclusion on these sets. In the language defined in this appendix and in full *FX*, there is no aliasing between region constants or region variables. As a result, we can assume that individual region constants and region variables correspond to disjoint sets of locations. This leads us to the definition of region inclusion given below.

**Definition**. The region description $\rho_1$ is included in the region description $\rho_2$, $\rho_1 \sqsubseteq \rho_2$, iff $FRC(\rho_1) \subseteq FRC(\rho_2) \quad \wedge \quad FDV(\rho_1) \subseteq FDV(\rho_2)$.

**Comment**. Since every region description is built up out of region constants and region variables, the set of region descriptions *modulo conversion* is isomorphic to $PowerSet(Rconst \cup Dvar)$, the set of all possible combinations of region constants and region variables. It follows that the region descriptions modulo conversion form a Boolean lattice.

## Effect Descriptions

As previously mentioned, an effect description corresponds to a set of `alloc`, `read`, and `write` effects on certain regions. We define the inclusion relation on effects to correspond to set inclusion on these sets of primitive effects. Note that under this interpretation, the effect constructors `alloc`, `read`, and `write` distribute over the region constructor `runion`: for example, the effect description (`alloc` (`runion` $r_1$ $r_2$)) corresponds to the same effect as the description (`maxeff` (`alloc` $r_1$) (`alloc` $r_2$)). This leads us to the definition of effect inclusion given below.

**Definition**. The inclusion relation $\sqsubseteq$ on effect descriptions is the partial order generated by the inference rules given below, where we identify descriptions that are equal modulo the distributive laws. We say that $\epsilon$ is a *subeffect* of $\epsilon'$ iff $\epsilon \sqsubseteq \epsilon'$.

$$\frac{\rho \sqsubseteq \rho'}{\begin{array}{rcl}(\textbf{alloc}\ \rho) & \sqsubseteq & (\textbf{alloc}\ \rho') \\ (\textbf{read}\ \rho) & \sqsubseteq & (\textbf{read}\ \rho') \\ (\textbf{write}\ \rho) & \sqsubseteq & (\textbf{write}\ \rho')\end{array}}$$

$$\frac{\epsilon_i \sqsubseteq \epsilon \text{ for all } 1 \leq i \leq n}{(\textbf{maxeff}\ \epsilon_1 \ldots \epsilon_n) \sqsubseteq \epsilon}$$

$$\frac{\epsilon \sqsubseteq \epsilon_i \text{ for some } 1 \leq i \leq n}{\epsilon \sqsubseteq (\textbf{maxeff}\ \epsilon_1 \ldots \epsilon_n)}$$

In other words, the effect constructors **alloc**, **read**, and **write** are *monotonic* with respect to description inclusion, and the effect constructor **maxeff** acts as the $n$-ary least upper bound operation.

**Comment.** Since every effect description is built up out of primitive effect descriptions (of the form $(\textbf{alloc}\ \rho)$, $(\textbf{read}\ \rho)$ or $(\textbf{write}\ \rho)$) and effect variables, the set of effect descriptions modulo conversion is isomorphic to $\mathit{PowerSet}((\{\textbf{alloc}, \textbf{read}, \textbf{write}\} \times (\mathit{Rconst} \cup \mathit{Dvar})) \cup \mathit{Dvar})$, the set of all possible combinations of primitive effects and effect variables. It follows that the effect descriptions modulo conversion form a Boolean lattice.

## Type Descriptions

As previously mentioned, a type description corresponds to a set of values. We define the inclusion relation on types to correspond to set inclusion on these sets of values.

**Definition.** The inclusion relation $\sqsubseteq$ on type descriptions is the partial order generated by the inference rules below, where we identify descriptions that are equal modulo the distributive laws and alpha-renaming. We say that $\tau$ is a *subtype* of $\tau'$ iff $\tau \sqsubseteq \tau'$.

The type inclusion inference rule for the type constructor subr reflects the fact that subr is *monotonic* in its effect and return type components, but *anti-monotonic* in its parameter type component. This follows from its interpretation as a generalization of the type constructor $\rightarrow$ in the typed lambda-calculus.

$$\frac{\epsilon \sqsubseteq \epsilon' \quad \wedge \quad \tau_1 \sqsupseteq \tau_1' \quad \wedge \quad \tau_2 \sqsubseteq \tau_2'}{(\mathbf{subr}\ \epsilon\ (\tau_1)\ \tau_2) \sqsubseteq (\mathbf{subr}\ \epsilon'\ (\tau_1')\ \tau_2')}$$

The rule for the type constructor poly reflects the fact that poly is monotonic in its return type component

$$\frac{\tau \sqsubseteq \tau'}{(\mathbf{poly}\ (d\ \kappa)\ \tau) \sqsubseteq (\mathbf{poly}\ (d\ \kappa)\ \tau')}$$

The rule for the type constructor `ref` reflects the fact that `ref` is monotonic in its region component, but neither monotonic nor anti-monotonic in its type component. As for the region component, monotonicity follows from the fact that the inclusion relation on region descriptions corresponds to set inclusion on the underlying sets of locations. As for the type component, the rule reflects the fact that in the presence of side-effects, neither $\tau \sqsubseteq \tau'$ nor $\tau \sqsupseteq \tau'$ implies (`ref` $\tau$ $\rho$) $\sqsubseteq$ (`ref` $\tau'$ $\rho$).

$$\frac{\rho \sqsubseteq \rho' \quad \wedge \quad \tau \simeq \tau'}{(\mathtt{ref}\ \tau\ \rho) \sqsubseteq (\mathtt{ref}\ \tau'\ \rho')}$$

To understand why `ref` is not monotonic in its type component, consider this. Computationally, a location is equivalent to a pair of subroutines, one for reading and one for writing. Thus, the type (`ref` $\tau$ $\rho$) is in some sense equivalent to a tuple of *two* types:

(`subr` (`read` $\rho$) (`unit`) $\tau$)
and
(`subr` (`write` $\rho$) ($\tau$) `unit`)

Since $\tau$ appears as the return type in the first type, (`ref` $\tau$ $\rho$) is a subtype of (`ref` $\tau'$ $\rho$) only if $\tau \sqsubseteq \tau'$. Yet, since $\tau$ appears as the parameter type in the second type, (`ref` $\tau$ $\rho$) is a subtype of (`ref` $\tau'$ $\rho$) only if $\tau \sqsupseteq \tau'$. It follows that (`ref` $\tau$ $\rho$) is neither monotonic nor anti-monotonic in $\tau$.

**Comment.** The type descriptions modulo conversion do not form a lattice, because the set is not closed under $\sqcup$ and $\sqcap$; for example, the type descriptions bool and (`ref` bool $r_1$) have neither an upper nor a lower bound.

## B.4 Kind, Type, and Effect Inference

Every well-formed description has a *kind*, which is one of **region**, **effect** or **type**. Similarly, every well-formed ordinary expression has both a *type* and an *effect* description. In this section we present a set of axioms and inference rules for determining the kind of a description and the type and effect descriptions of an ordinary expression.

### Kinds

Since a description may have free description variables, the kind of a description is defined in the context of a *kind assignment*, which is a partial function with signature *Dvar* → *Kind* that maps description variables to their kinds. We use *Kas* to denote the set of kind assignments and $B$ to denote individual kind assignments. We use the notation $f[x \mapsto y]$ to denote the function similar to $f$ except that it maps $x$ to $y$. The relation *has kind*, or :: , on $(Desc \times Kas) \times Kind$ gives the kind, if any, of a tuple consisting of a description and a kind assignment.

**Definition.** The relation *has kind* is the least relation consistent with the axioms and inference rules below. The axioms state that every region constant has kind **region**, every type constant has kind **type**, and every description variable $d$ in the domain of $B$ has kind $B(d)$.

$$\begin{array}{lcl} \langle r, B\rangle & :: & \textbf{region} \\ \langle t, B\rangle & :: & \textbf{type} \\ \langle d, B\rangle & :: & B(d) \end{array}$$

The kind inference rule for region descriptions states that the **runion** of one or more descriptions of kind **region** also has kind **region**.

$$\frac{\langle \rho_i, B\rangle :: \textbf{region} \quad \text{for all } 1 \le i \le n}{\langle (\textbf{runion}\ \ \rho_1 \ldots \rho_n), B\rangle :: \textbf{region}}$$

There are two kind inference rules for effect descriptions. The first rule states that if $\rho$ has kind **region**, then the effect descriptions (**alloc** $\rho$), (**read** $\rho$) and (**write** $\rho$) all have kind **effect**.

$$\frac{\langle \rho, B\rangle :: \textbf{region}}{\begin{array}{rcl} \langle (\textbf{alloc}\ \rho), B\rangle & :: & \textbf{effect} \\ \langle (\textbf{read}\ \rho), B\rangle & :: & \textbf{effect} \\ \langle (\textbf{write}\ \rho), B\rangle & :: & \textbf{effect} \end{array}}$$

The second rule states that the **maxeff** of zero or more descriptions of kind **effect** also has kind **effect**. In particular, the effect description (**maxeff**) or **pure** has kind **effect**.

$$\frac{\langle \epsilon_i, B\rangle :: \textbf{effect} \quad \text{for all } 1 \leq i \leq n}{\langle(\textbf{maxeff}\ \epsilon_1 \ldots \epsilon_n), B\rangle :: \textbf{effect}}$$

Finally, there are three kind inference rules for type descriptions. The first two rules are direct adaptations of the corresponding rules in the second-order typed lambda-calculus, changed only to handle the effect component of the **subr** and **poly** type descriptions.

$$\frac{\begin{array}{rcl} \langle \epsilon, B\rangle & :: & \textbf{effect} \\ \langle \tau_1, B\rangle & :: & \textbf{type} \\ \langle \tau_2, B\rangle & :: & \textbf{type} \end{array}}{\langle(\textbf{subr}\ \epsilon\ (\tau_1)\ \tau_2), B\rangle :: \textbf{type}}$$

$$\frac{\begin{array}{rcl} \langle \tau, B[d \mapsto \kappa]\rangle & :: & \textbf{type} \\ \langle \epsilon, B[d \mapsto \kappa]\rangle & :: & \textbf{effect} \end{array}}{\langle(\textbf{poly}\ (d\ \kappa)\ \tau), B\rangle :: \textbf{type}}$$

The rule for **ref** type descriptions simply states that if $\rho$ has kind **ref** and $\tau$ has kind **type**, then (**ref** $\tau$ $\rho$) has kind **type**.

$$\frac{\begin{array}{rcl} \langle \rho, B\rangle & :: & \textbf{region} \\ \langle \tau, B\rangle & :: & \textbf{type} \end{array}}{\langle(\textbf{ref}\ \tau\ \rho), B\rangle :: \textbf{type}}$$

**Definition.** A tuple $\langle \delta, B\rangle$ is *well-formed*, $\mathcal{WF}(\langle \delta, B\rangle)$, iff it has a kind, *i.e.* iff $\langle \delta, B\rangle :: \kappa$ for some kind $\kappa$.

**Definition.** A closed description $\delta$ is *well-formed*, $\mathcal{WF}(\delta)$, iff it has a kind under the empty kind assignment, *i.e.* iff $\langle \delta, \phi\rangle :: \kappa$ for some $\kappa$. If $\delta$ is closed and well-formed and $\langle \delta, \phi\rangle :: \kappa$, we write $\delta :: \kappa$ and say that $\delta$ has kind $\kappa$.

**Fact.** If the tuple $\langle \delta, B\rangle$ is well-formed, then its kind is *unique*; in other words, if $\langle \delta, B\rangle :: \kappa_1$ and $\langle \delta, B\rangle :: \kappa_2$ then $\kappa_1 = \kappa_2$. It follows that the relation *has kind* is a partial function.

## The Types and Effects of Expressions

In this section we give a set of axioms and rules for determining the type and effect description of an ordinary expression. Because an ordinary expression may have free ordinary and description variables, its type and effect description are defined in the context of both a kind assignment and a *type assignment*, which is a partial function with signature *Var* → *Type* that maps ordinary variables to their type descriptions. We use *Tas* to denote the set of type assignments and $A$ to denote individual type assignments.

The relation *has type*, or $:$ , on $(exp \times Tas \times Kas) \times Type$ gives the type description, if any, of a tuple $\langle e, A, B\rangle$ consisting of an ordinary expression, a type assignment, and a kind assignment. Likewise, the relation *has effect*, or $!$ , on $(exp \times Tas \times Kas) \times Effect$ gives the (unmasked) effect description, if any, of such a tuple. Effect masking is described below.

**Definition.** The relations *has type* and *has effect* are the least relations consistent with the axioms and inference rules below.

The type axioms, below, state that `#u` has type `unit`; the Booleans `#t` and `#f` have type `bool`; and every ordinary variable $x$ in the domain of $A$ has type $A(x)$.

$$\begin{array}{rcl} \langle \texttt{\#u}, A, B\rangle & : & \texttt{unit} \\ \langle b, A, B\rangle & : & \texttt{bool} \\ \langle x, A, B\rangle & : & A(x) \end{array}$$

The effect axioms, below, state that every ordinary constant or variable and every abstraction expression, whether ordinary or polymorphic, has effect `pure`, regardless of the type or kind assignment.

$$\begin{array}{c} \langle v, A, B\rangle \ ! \ \texttt{pure} \\ \langle x, A, B\rangle \ ! \ \texttt{pure} \end{array}$$

The type inference rule for ordinary abstraction is a generalization of the corresponding rule in the typed lambda-calculus. The main difference is that the *effect* description of the body of the `lambda` expression is incorporated into the *type* description of the expression itself. This reflects the fact that the side-effects (if any) of the body do not take place when the `lambda` expression is evaluated, but when the subroutine is applied. Note that the *effect* description of a `lambda` expression is always `pure`, by the corresponding effect axiom.

$$\frac{\begin{array}{rcl} \langle e, A[x \mapsto \tau_1], B\rangle & ! & \epsilon \\ \langle e, A[x \mapsto \tau_1], B\rangle & : & \tau_2 \end{array}}{\langle (\texttt{lambda}\ (x\ \tau_1)\ e), A, B\rangle : (\texttt{subr}\ \epsilon\ (\tau_1)\ \tau_2)}$$

The rule for application is a generalization of the corresponding rule in the typed lambda-calculus. The main difference is that the effect description embedded in the *type* of the subroutine is incorporated into the *effect* description of the application as a whole. This reflects the fact that the side-effects (if any) of the body of the subroutine take place when the subroutine is applied. Note that the type description of the actual parameter need not match that of the formal parameter exactly, but must be included in it.

$$\frac{\begin{array}{rcl} \langle e_1, A, B\rangle & : & (\mathtt{subr}\ \epsilon\ (\tau_1)\ \tau_2) \\ \langle e_2, A, B\rangle & : & \tau \wedge \tau \sqsubseteq \tau_1 \\ \langle e_1, A, B\rangle & ! & \epsilon_1 \\ \langle e_2, A, B\rangle & ! & \epsilon_2 \end{array}}{\begin{array}{rcl} \langle (e_1\ e_2), A, B\rangle & : & \tau_2 \\ \langle (e_1\ e_2), A, B\rangle & ! & (\mathtt{maxeff}\ \epsilon_1\ \epsilon_2\ \epsilon) \end{array}}$$

The rule for polymorphic abstraction is a generalization of the corresponding rule in the second-order typed lambda-calculus. The main difference is that the *effect* description of the body of the **plambda** expression must be **pure**. Note that this rule ensures that the free description variables of the types of the free variables of the body are not captured by the bound description variable. Also, the *effect* description of a **plambda** expression is always **pure**, by the corresponding effect axiom.

$$\frac{\begin{array}{rcl} \langle e, A, B[d \mapsto \kappa]\rangle & : & \tau \\ \langle e, A, B[d \mapsto \kappa]\rangle & ! & \mathtt{pure} \\ x \in FV(e) & \Rightarrow & d \notin FDV(A(x)) \end{array}}{\langle (\mathtt{plambda}\ (d\ \kappa)\ e), A, B\rangle : (\mathtt{poly}\ (d\ \kappa)\ \tau)}$$

The rule for projection is a generalization of the corresponding rule in the second-order typed lambda-calculus. This rule also enforces the anti-aliasing provision of the language: it ensures that if the actual parameter is a region description, then the expression as a whole is not well-typed unless the actual parameter is disjoint from the free region variables and region constants of the type of the operator. This ensures that the application does not create aliasing between region constants and/or region variables.

$$\frac{\begin{array}{rcl} \langle e, A, B\rangle & : & \tau = (\texttt{poly}\ (d\ \kappa)\ \tau') \\ \langle e, A, B\rangle & ! & \epsilon \\ \langle \delta, B\rangle & :: & \kappa \\ \kappa = \texttt{region} & \Rightarrow & \begin{cases} FRC(\tau) \cap FRC(\delta) = \phi \\ FDV(\tau) \cap FDV(\delta) = \phi \end{cases} \end{array}}{\begin{array}{rcl} \langle (\texttt{proj}\ e\ \delta), A, B\rangle & : & \tau'[\delta/d] \\ \langle (\texttt{proj}\ e\ \delta), A, B\rangle & ! & \epsilon \end{array}}$$

The rule for conditional expressions ensures that the first subexpression has type description `bool`, and that the remaining two subexpressions have type descriptions whose maximum exists. When this is the case, the type description of the `if` expression is this maximum, and its effect description is the maximum bound of the effect descriptions of its subexpressions, reflecting the fact that evaluating an `if` expression involves evaluating some subset of its subexpressions.

$$\frac{\begin{array}{rr} \langle e_1, A, B\rangle : \texttt{bool} & \langle e_1, A, B\rangle\ !\ \epsilon_1 \\ \langle e_2, A, B\rangle : \tau_2 & \langle e_2, A, B\rangle\ !\ \epsilon_2 \\ \langle e_3, A, B\rangle : \tau_3 & \langle e_3, A, B\rangle\ !\ \epsilon_3 \\ \tau_2 \sqcup \tau_3 = \tau\ \ \wedge & (\tau = \tau_2\ \ \vee\ \ \tau = \tau_3) \end{array}}{\begin{array}{rcl} \langle (\texttt{if}\ e_1\ e_2\ e_3), A, B\rangle & : & \tau \\ \langle (\texttt{if}\ e_1\ e_2\ e_3), A, B\rangle & ! & (\texttt{maxeff}\ \epsilon_1\ \epsilon_2\ \epsilon_3) \end{array}}$$

The rule for sequencing ordinary expressions is remarkably simple: provided that each subexpression has a type and effect description, the type description of a `begin` expression is the same as that of the last subexpression, and the effect description is the least upper bound of the effect descriptions of the subexpressions.

$$\frac{\begin{array}{rl} \langle e_i, A, B\rangle : \tau_i & \text{for all } 1 \le i \le n \\ \langle e_i, A, B\rangle\ !\ \epsilon_i & \text{for all } 1 \le i \le n \end{array}}{\begin{array}{rcl} \langle (\texttt{begin}\ e_1 \ldots e_n), A, B\rangle & : & \tau_n \\ \langle (\texttt{begin}\ e_1 \ldots e_n), A, B\rangle & ! & (\texttt{maxeff}\ \epsilon_1 \ldots \epsilon_n) \end{array}}$$

The rule for effect/type assertion allows an expression of some particular effect and type to be regarded as though it had a larger effect and type.

$$\frac{\begin{array}{rcl} \langle \tau', B\rangle & :: & \texttt{type} \\ \langle \epsilon', B\rangle & :: & \texttt{effect} \\ \langle e, A, B\rangle & : & \tau \\ \langle e, A, B\rangle & ! & \epsilon \\ \tau \sqsubseteq \tau' & \wedge & \epsilon \sqsubseteq \epsilon' \end{array}}{\begin{array}{rcl} \langle (\texttt{the}\ \epsilon'\ \tau'\ e), A, B\rangle & : & \tau' \\ \langle (\texttt{the}\ \epsilon'\ \tau'\ e), A, B\rangle & ! & \epsilon' \end{array}}$$

The remaining three rules deal with the expressions for allocating, reading, and writing locations. The rule for the **new** expression can be read as follows: provided that $\rho$ has kind **region**, $\tau$ has kind **type**, and $e$ has a type and effect description, and provided that the type description of $e$ is included in $\tau$, the type description of the expression as a whole is (**ref** $\tau$ $\rho$), and the effect description of the expression is the least upper bound of the effect description of $e$ and the primitive effect description (**alloc** $\rho$).

$$\frac{\begin{array}{rcl} \langle \rho, B\rangle & :: & \texttt{region} \\ \langle \tau, B\rangle & :: & \texttt{type} \\ \langle e, A, B\rangle : \tau' & \wedge & \tau' \sqsubseteq \tau \\ \langle e, A, B\rangle & ! & \epsilon \end{array}}{\begin{array}{rcl} \langle (\texttt{new}\ \tau\ \rho\ e), A, B\rangle & : & (\texttt{ref}\ \tau\ \rho) \\ \langle (\texttt{new}\ \tau\ \rho\ e), A, B\rangle & ! & (\texttt{maxeff}\ \epsilon\ (\texttt{alloc}\ \rho)) \end{array}}$$

The rule for the **get** expression can be read as follows: if the type description of $e$ is (**ref** $\tau$ $\rho$), then the type description of the expression as a whole is $\tau$, and its effect description is the least upper bound of the effect description of $e$ and the primitive effect description (**read** $\rho$).

$$\frac{\begin{array}{rcl} \langle e, A, B\rangle & : & (\texttt{ref}\ \tau\ \rho) \\ \langle e, A, B\rangle & ! & \epsilon \end{array}}{\begin{array}{rcl} \langle (\texttt{get}\ e), A, B\rangle & : & \tau \\ \langle (\texttt{get}\ e), A, B\rangle & ! & (\texttt{maxeff}\ \epsilon\ (\texttt{read}\ \rho)) \end{array}}$$

The rule for the **set** expression can be read as follows: provided that the type description of $e_1$ is (**ref** $\tau$ $\rho$) such that the type description of $e_2$ is included in $\tau$, the type description of the expression as a whole is **unit**, and its effect description is the least upper bound of the effect descriptions of $e_1$

and $e_2$ and the primitive effect description (**write** $\rho$).

$$\frac{\begin{array}{lcl} \langle e_1, A, B\rangle & : & (\mathtt{ref}\ \tau\ \rho) \\ \langle e_2, A, B\rangle & : & \tau' \wedge \tau' \sqsubseteq \tau \\ \langle e_1, A, B\rangle & ! & \epsilon_1 \\ \langle e_2, A, B\rangle & ! & \epsilon_2 \end{array}}{\begin{array}{lcl} \langle(\mathtt{set}\ e_1\ e_2), A, B\rangle & : & \mathtt{unit} \\ \langle(\mathtt{set}\ e_1\ e_2), A, B\rangle & ! & (\mathtt{maxeff}\ \epsilon_1\ \epsilon_2\ (\mathtt{write}\ \rho)) \end{array}}$$

**Definition.** A tuple $\langle e, A, B\rangle$ is *well-formed*, $\mathcal{WF}(\langle e, A, B\rangle)$, iff it has a type description, *i.e.* iff $\langle e, A, B\rangle : \tau$ for some $\tau$.

**Definition.** A closed expression $e$ is *well-formed*, $\mathcal{WF}(e)$, iff it has a type description under the empty type and kind assignments, *i.e.* iff $\langle e, \phi, \phi\rangle : \tau$ for some $\tau$. If $e$ is closed and well-formed and $\langle e, \phi, \phi\rangle : \tau$, we write $e : \tau$ and say that $e$ has type $\tau$; similarly, if $e$ is closed and well-formed and $\langle e, \phi, \phi\rangle\ !\ \epsilon$, we will write $e\ !\ \epsilon$ and say that $e$ has effect $\epsilon$.

**Fact.** Every well-formed tuple has an effect description; in other words, if $\langle e, A, B\rangle : \tau$ for some $\tau$ then $\langle e, A, B\rangle\ !\ \epsilon$ for some $\epsilon$.

**Fact.** If the tuple $\langle e, A, B\rangle$ is well-formed, then its type and effect descriptions are themselves well-formed and of kind **type** and **effect** respectively, provided that the type descriptions in $A$ of the free variables of $e$ are well-formed. In other words, if $\langle A(x), B\rangle$ is well-formed for each $x \in FV(e)$, then if $\langle e, A, B\rangle : \tau$ and $\langle e, A, B\rangle\ !\ \epsilon$ then $\langle \tau, B\rangle :: \mathtt{type}$ and $\langle \epsilon, B\rangle :: \mathtt{effect}$.

**Fact.** If the tuple $\langle e, A, B\rangle$ is well-formed, then its type and effect descriptions are *unique* modulo conversion; in other words, if $\langle e, A, B\rangle : \tau_1$ and $\langle e, A, B\rangle : \tau_2$ then $\tau_1 \simeq \tau_2$, and likewise if $\langle e, A, B\rangle\ !\ \epsilon_1$ and $\langle e, A, B\rangle\ !\ \epsilon_2$ then $\epsilon_1 \simeq \epsilon_2$. It follows that the relations *has type* and *has effect* are partial functions (modulo description conversion).

## Effect Masking

Effects that are not observable outside of a given expression can be *masked* by the type and effect system. To this end, all expressions are subjected to effect masking according to the following rule:

> If the expression has effects on some region identifier $d$, and $d$ does not appear free in the type of any free variable of the expression, then any read or write effect on $d$ is masked; furthermore,

if $d$ does not appear free in the type of the expression, then any `alloc` effect on $d$ is masked too.

In order to define effect masking formally, we introduce the pseudo-region $o$. Conceptually, $o$ corresponds to an empty region, *i.e.* a region to which no locations belong. Thus, (`runion` $o$ $\rho$) is convertible with $\rho$ (for all $\rho$), and the effects (`alloc` $o$), (`read` $o$) and (`write` $o$) are convertible with `pure`. The pseudo-region $o$ can be regarded as the bottom of the region lattice. We can now give the formal effect masking inference rules:

$$\frac{\begin{array}{rcl} \langle e,A,B\rangle & : & \tau \\ \langle e,A,B\rangle & ! & \epsilon \\ \langle d,B\rangle & :: & \mathtt{region} \\ x \in FV(e) & \Rightarrow & d \notin FV(A(x)) \\ d & \notin & FV(\tau) \end{array}}{\langle e,A,B\rangle \;\; ! \;\; \epsilon[o/d]}$$

$$\frac{\begin{array}{rcl} \langle e,A,B\rangle & : & \tau \\ \langle e,A,B\rangle & ! & \epsilon \\ \langle d,B\rangle & :: & \mathtt{region} \\ x \in FV(e) & \Rightarrow & d \notin FV(A(x)) \end{array}}{\langle e,A,B\rangle \;\; ! \;\; (\mathtt{maxeff}\ \epsilon[o/d]\ \ (\mathtt{alloc}\ d))}$$

Due to effect masking, an expression may have many effects: one that has not benefited from effect masking, one in which nothing remains to be masked, and any number that are somewhere in between. To deal with effect masking, any formula of the form $\langle e,A,B\rangle \; ! \; \epsilon$ in a *premise* of a type and effect inference rule must be interpreted as meaning that $\epsilon$ is the *least* effect of $\langle e,A,B\rangle$. With this interpretation, every well-formed tuple $\langle e,A,B\rangle$ has a type description and a least effect description that are unique modulo conversion.

## B.5 Standard Semantics

The standard semantics of the language are based on the standard rewrite rules for the second-order typed lambda-calculus; in particular, the semantics of application as well as projection are expressed in terms of beta-substitution. Side-effects are modeled using a *store* that maps locations to values. To avoid the complications that arise when a computation runs out of unused locations, we define a store to be a *finite* function with signature $Loc \rightarrow Val$ that maps locations to values. Since the number of locations is infinite and every finite computation allocates only a finite number of locations, this definition ensures that a computation never runs out of unused locations. We use *Store* to denote the set of stores and $\sigma$ to denote individual stores.

Because of side-effects, the order of subexpression evaluation is crucial to the semantics of the language. As a result, the rewrite rules are directional — hence, from now on, we use the term *reduction* rather than *rewriting*.

To avoid over-specification, we have defined the standard semantics so that new locations, when needed, are chosen nondeterministically. This gives the language implementation a great deal of flexibility, which is essential to permit such optimizations as code motion, common subexpression elimination, and dead code elimination.

## Locations

Before we can describe the semantics, we must define what we mean by locations. Formally, locations are a countably infinite set of constants:

| | | |
|---|---|---|
| $Loc =$ | $\{l_1, l_2, \ldots\}$ | – locations ($l$) |
| $Const =$ | $\ldots$ | – ordinary constants ($c$) |
| | $Loc$ | – the locations |

A location can be *tagged* with a region description and a type description. The region tag of a location indicates to what region the location belongs, and the type tag of a location indicates what types of values the location may contain. Specifically, a location tagged with a region description $\rho$ belongs to the region $\rho$, and a location tagged with a type description $\tau$ may only contain values whose type is a subtype of $\tau$. These descriptions ought to be *closed*; tags that contain free description variables are meaningless.

We write $R(l)$ for the region tag of the location $l$ and $T(l)$ for its type tag. Moreover, we write $l_{\rho,\tau}$ to indicate that $R(l_{\rho,\tau}) = \rho$ and $T(l_{\rho,\tau}) = \tau$.

Every closed region description $\rho$ corresponds to a nonempty set of region constants, namely $FRC(\rho)$. If $\rho$ is a region constant, then the location $l_{\rho,\tau}$ belongs to the region corresponding to that region constant. If $\rho$ is a `runion` of several region constants, then the location $l_{\rho,\tau}$ belongs to the union of the corresponding regions. This situation reflects either *uncertainty* or *indifference* about the region constant to which the location actually belongs.

It is convenient to define the free locations of an ordinary expression, $FL(e)$. Since locations are constants, the definition of $FL$ is trivial: all the locations that occur in an ordinary expression are free.

**Definition.** The free locations of an ordinary expression are given by the function $FL : \mathbf{exp} \rightarrow \mathit{PowerSet}(Loc)$.

Since locations are constants and therefore ordinary expressions, we must define their free ordinary and description variables, their free region constants, their types, and their effects. The first few are easy: since locations are constants, they have neither free ordinary variables nor free description variables. However, because of its region and type tag, a location may have free region constants:

$$FRC(l_{\rho,\tau}) = FRC(\rho) \cup FRC(\tau)$$

Because locations are constants, their effect is pure. Finally, the type of a location is a `ref` type whose region and type parameters are equal to the region and type tags of the location:

$$\langle l_{\rho,\tau}, A, B\rangle : (\mathtt{ref}\ \tau\ \rho)$$

## Stores and States

The state of a computation consists of two components: an ordinary expression, which indicates the computation that remains to be performed, and a *store*, which maps locations to values.

**Definition.** A *store* is a finite function $\sigma : Loc \rightarrow Val$ that maps locations to values. We use *Store* to denote the set of stores and $\sigma$ to denote individual stores.

**Definition.** A *state* is a tuple $\langle e, \sigma\rangle \in (\mathbf{exp} \times Store)$. We use *State* to denote the set of states and $\theta$ to denote individual states.

## Reduction

The reduction relation $\xRightarrow{\text{red}}$ on $(State \times State)$ is defined by a set of reduction axioms and a set of reduction inference rules. The reduction axioms show how to reduce an ordinary expression when certain of its subexpressions have been reduced to values; the reduction inference rules show how to reduce an ordinary expression to which none of the reduction axioms applies by reducing one of its subexpressions.

A *value* cannot be reduced; in other words, for all $v$ and $\sigma$ there is no $\theta$ such that $\langle v, \sigma\rangle \xRightarrow{\text{red}} \theta$. We make extensive use of this fact to ensure that subexpressions are evaluated in left-to-right applicative order. For example, the reduction axiom for ordinary application (shown below) is applicable only when the operator is a `lambda` expression, which is a value, and the operand is a value as well. This technique is used throughout to keep the reduction axioms and inference rules from being invoked prematurely.

The first two axioms, which deal with application and projection, are adapted directly from the second-order typed lambda calculus. Note that the store is not affected.

$$\begin{aligned} \langle((\texttt{lambda}\ (x\ \tau)\ e)\ v), \sigma\rangle &\xRightarrow{\text{red}} \langle e[v/x], \sigma\rangle \\ \langle((\texttt{proj}\ (\texttt{plambda}\ (d\ \kappa)\ e)\ \delta), \sigma\rangle &\xRightarrow{\text{red}} \langle e[\delta/d], \sigma\rangle \end{aligned}$$

The axiom for ordinary application may entail duplication of the actual parameter. This does not cause any problems, despite the possibility of side-effects, because the actual parameter is a *value*, which cannot be further reduced.

The next set of axioms, which deal with conditional and sequential evaluation, should be more or less self-explanatory.

$$\begin{aligned} \langle(\texttt{if \#t}\ e_2\ e_3), \sigma\rangle &\xRightarrow{\text{red}} \langle e_2, \sigma\rangle \\ \langle(\texttt{if \#f}\ e_2\ e_3), \sigma\rangle &\xRightarrow{\text{red}} \langle e_3, \sigma\rangle \\ \langle(\texttt{begin}\ v), \sigma\rangle &\xRightarrow{\text{red}} \langle v, \sigma\rangle \\ \langle(\texttt{begin}\ v\ e_1 \ldots e_n), \sigma\rangle &\xRightarrow{\text{red}} \langle(\texttt{begin}\ e_1 \ldots e_n), \sigma\rangle \quad (n > 0) \\ \langle(\texttt{the}\ \epsilon\ \tau\ v), \sigma\rangle &\xRightarrow{\text{red}} \langle v, \sigma\rangle \end{aligned}$$

The remaining axioms deal with the allocating, reading, and writing of locations. In what follows, we use the notation $\sigma[l \mapsto v]$ to denote the store that is identical to $\sigma$ except that it maps $l$ to $v$, while simultaneously expressing the fact that $l$ is not bound in $\sigma$. We use the symbol "$-$" to denote an undefined value, so that $\sigma[l \mapsto -]$ denotes the store $\sigma$ while expressing the fact that $l$ is not bound in $\sigma$.

The axiom for the **new** expression can be read as follows. To reduce the expression (**new** $\tau$ $\rho$ $v$), choose any location $l$ of type $\tau$ in the region $\rho$ that is not bound in the store. By the definition of a store, such a location exists iff $\tau$ and $\rho$ are closed. Once a suitable location $l$ has been chosen, simply bind $l$ to $v$ in the store, and replace the expression by the value $l$.

$$\langle(\textbf{new}\ \tau\ \rho\ v), \sigma[l_{\rho,\tau} \mapsto -]\rangle \overset{\text{red}}{\Longrightarrow} \langle l, \sigma[l \mapsto v]\rangle$$

This axiom represents a non-deterministic reduction: unlike the other axioms, this axiom permits a state to reduce (in one step) to a countably infinite number of states, differing only in their choice of the new location. The course of a computation is not affected by the choice of new locations.

To reduce the expression (**get** $l$), where $l$ is bound to $v$ in the store, simply replace the expression by the value $v$.

$$\langle(\textbf{get}\ l), \sigma[l \mapsto v]\rangle \overset{\text{red}}{\Longrightarrow} \langle v, \sigma[l \mapsto v]\rangle$$

To reduce the expression (**set** $l$ $v$), where $l$ is bound in the store, simply bind $l$ to $v$ in the store and replace the expression by the value #u.

$$\langle(\textbf{set}\ l\ v), \sigma[l \mapsto v']\rangle \overset{\text{red}}{\Longrightarrow} \langle \#\text{u}, \sigma[l \mapsto v]\rangle$$

This concludes the set of reduction axioms. Note that each of these axioms is applicable only when certain subexpressions of the outermost ordinary expression are *values*. The reduction inference rules, which are given below, show how to reduce an ordinary expression to which none of the reduction axioms applies by reducing one of its subexpressions. There are quite a few of these rules: two for application (one for the operator subexpression and one for the operand), one for projection (since the operand is a description, which does not need to be reduced), one each for `if`, **begin**, and the one each for **new** and **get**, and two for **set**. The rules are all structured so that subexpressions are evaluated in left-to-right, applicative order.

$$\frac{\langle e_1, \sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e_1', \sigma'\rangle}{\langle(e_1\ e_2), \sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(e_1'\ e_2), \sigma'\rangle}$$

$$\frac{\langle e_2, \sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e_2', \sigma'\rangle}{\langle(v_1\ e_2), \sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(v_1\ e_2'), \sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e',\sigma'\rangle}{\langle(\mathbf{proj}\ e\ \delta),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{proj}\ e'\ \delta),\sigma'\rangle}$$

$$\frac{\langle e_1,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e_1',\sigma'\rangle}{\langle(\mathbf{if}\ e_1\ e_2\ e_3),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{if}\ e_1'\ e_2\ e_3),\sigma'\rangle}$$

$$\frac{\langle e_1,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e_1',\sigma'\rangle}{\langle(\mathbf{begin}\ e_1\ e_2\ldots e_n),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{begin}\ e_1'\ e_2\ldots e_n),\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e',\sigma'\rangle}{\langle(\mathbf{the}\ \epsilon\ \tau\ e),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{the}\ \epsilon\ \tau\ e'),\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e',\sigma'\rangle}{\langle(\mathbf{new}\ \tau\ \rho\ e),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{new}\ \tau\ \rho\ e'),\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e',\sigma'\rangle}{\langle(\mathbf{get}\ e),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{get}\ e'),\sigma'\rangle}$$

$$\frac{\langle e_1,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e_1',\sigma'\rangle}{\langle(\mathbf{set}\ e_1\ e_2),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{set}\ e_1'\ e_2),\sigma'\rangle}$$

$$\frac{\langle e_2,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle e_2',\sigma'\rangle}{\langle(\mathbf{set}\ v\ e_2),\sigma\rangle \overset{\text{red}}{\Longrightarrow} \langle(\mathbf{set}\ v\ e_2'),\sigma'\rangle}$$

## Stuck States

**Definition.** A state $\langle e,\sigma\rangle$ is *stuck* iff $e$ is not a value and the state cannot be reduced, *i.e.* iff $e \notin \mathit{Val}$ and there is no $\theta$ such that $\langle e,\sigma\rangle \overset{\text{red}}{\Longrightarrow} \theta$.

A comparison of the grammar of ordinary expressions on the one hand and the reduction axioms on the other hand yields a list of the different sorts of stuck states. If the state $\langle e,\sigma\rangle$ is stuck, then either $e$ contains a subexpression that is stuck, or $e$ is one of the following:

- a variable
- $(v_1\ v_2)$ where $v_1$ is not an ordinary subroutine
- $(\texttt{proj}\ v\ \delta)$ where $v$ is not a polymorphic value
- $(\texttt{if}\ v\ e_2\ e_3)$ where $v$ is not a Boolean
- $(\texttt{new}\ \tau\ \rho\ v)$ where $\rho$ and $\tau$ are not closed and of kind `region` and `type` respectively
- $(\texttt{get}\ v)$ where $v$ is not a location
- $(\texttt{get}\ l)$ where $l$ is not bound in the store
- $(\texttt{set}\ v_1\ v_2)$ where $v_1$ is not a location
- $(\texttt{set}\ l\ v)$ where $l$ is not bound in the store

These expressions can be divided into various categories: attempts to use undeclared variables, attempts to use uninitialized locations, type errors, and kind errors.

**Fact.** If the state $\langle e, \sigma \rangle$ is stuck, then $e$ must either be ill-formed or contain some location whose contents is either undefined or of the wrong type.

In the next section, we show that reduction of a well-formed state never gets stuck. In particular, this implies that type checking prevents run-time type errors and attempts to use uninitialized locations.

## B.6 Types Revisited

In this section we present several properties of the language that have to do with types. We begin by generalizing the notion of well-formed ordinary expression to that of a well-formed state. We can then verify that a well-formed state is not stuck. Moreover, we can prove that reduction of a well-formed state yields another well-formed state whose type and effect are at most those of the original state. Finally, we show that reduction of a well-formed state never gets stuck. The results of this section do not deal with effect masking.

## Type Soundness

In this section we prove that reduction of a well-formed state yields another well-formed state and preserves or decreases the type and effect descriptions of the state. As an intermediate step, we introduce the notion of a well-formed *store*. Informally, a store is well-formed iff all the values in the store are well-formed and are of the right type. More formally, we have the following definitions.

**Definition.** A store is *well-formed*, $\mathcal{WF}(\sigma)$, iff every value in the store is well-formed and has a type description that is included in the type of its location. In other words,

$$\mathcal{WF}(\sigma) \quad \Leftrightarrow \quad (\sigma(l_{\rho,\tau}) = v \;\Rightarrow\; v : \tau' \wedge \tau' \sqsubseteq \tau)$$

We also introduce the notion of a *consistent* state. Informally, a state is consistent iff all the locations that occur in the state are bound in its store component.

**Definition.** A location $l$ *occurs* in a store $\sigma$ iff either $\sigma(l) = v$ for some $v$, or $l$ occurs in a value $\sigma(l')$ for some $l'$. The locations that occur in a store are given by the function $FL_{store}$, which is formally defined below.

$$FL_{store}(\sigma) = \bigcup_{l \in Domain(\sigma)} (\{\, l \,\} \cup FL(\sigma(l)))$$

**Definition.** A location $l$ occurs in a state $\langle e, \sigma\rangle$ iff it occurs in the expression component $e$ or the store component $\sigma$. The locations that occur in a state are given by the function $FL_{state}$, which is formally defined below.

$$FL_{state}(\langle e,\sigma\rangle) \;= FL(e) \cup FL_{store}(\sigma)$$

**Definition.** A state is *consistent*, $Cons(\langle e,\sigma\rangle)$, iff every location that occurs in the state is bound in the store, *i.e.* iff $l \in FL(\langle e,\sigma\rangle)$ implies that $\sigma(l) = v$ for some $v$.

We can now define what constitutes a well-formed *state*.

**Definition.** A state $\langle e,\sigma\rangle$ is *well-formed*, $\mathcal{WF}(\langle e,\sigma\rangle)$, iff it is consistent and $e$ and $\sigma$ are both well-formed. In other words,

$$\mathcal{WF}(\langle e,\sigma\rangle) \quad \Leftrightarrow \quad Cons(\langle e,\sigma\rangle) \;\wedge\; \mathcal{WF}(e) \;\wedge\; \mathcal{WF}(\sigma)$$

If $\langle e, \sigma \rangle$ is well-formed and $e : \tau$, we write $\langle e, \sigma \rangle : \tau$ and say that $\langle e, \sigma \rangle$ has type $\tau$; similarly, if $\langle e, \sigma \rangle$ is well-formed and $e\ !\ \epsilon$, we write $\langle e, \sigma \rangle\ !\ \epsilon$ and say that $\langle e, \sigma \rangle$ has effect $\epsilon$.

Since a *program*, by definition, contains no locations, the state $\langle e, \phi \rangle$ is well-formed for any well-formed program $e$.

We can now express the type soundness claim. It is a generalization of the type soundness theorem (or subject reduction lemma) of the second-order typed lambda-calculus, which states that reduction of a well-typed ordinary expression yields another well-typed ordinary expression of the same type.

The claim presented here is more general than the type soundness theorem of the lambda-calculus in three respects: side-effects, description inclusion, and effect descriptions. To deal with side-effects, the claim has been generalized from *ordinary expressions* to *states*. To deal with description inclusion, the claim has been relaxed so that the reduction of a state of type $\tau$ may yield a state of *any* type $\tau' \sqsubseteq \tau$. Finally, to deal with effect descriptions, the following claim about effect descriptions has been added: the reduction of a state with effect $\epsilon$ must yield a state of any effect $\epsilon' \sqsubseteq \epsilon$.

**Claim**: (Type Soundness) Reduction of a well-formed state yields another well-formed state, preserves or decreases the type and effect descriptions of the state, and preserves or increases the set of locations bound in the store.

$$\begin{array}{ccc} \mathcal{WF}(\langle e, \sigma \rangle) & & \mathcal{WF}(\langle e', \sigma' \rangle) \\ e : \tau & \Rightarrow & e' : \tau' \text{ where } \tau' \sqsubseteq \tau \\ e\ !\ \epsilon & & e'\ !\ \epsilon' \text{ where } \epsilon' \sqsubseteq \epsilon \\ \langle e, \sigma \rangle \overset{red}{\Longrightarrow} \langle e', \sigma' \rangle & & \mathit{Domain}(\sigma') \supseteq \mathit{Domain}(\sigma) \end{array}$$

**Lemma**: A well-formed state is not stuck.

**Corollary**: (Static Typing) Reduction of a well-formed state never gets stuck; in particular, reduction of a well-formed state never encounters a type error, a kind error, or an uninitialized location.

## B.7 Effects Revisited

In this section we present several properties of the language that have to do with effects. The main property we show is that the actual side-effect of reducing a well-formed state is equal to at most the syntactic side-effect of the original state. This property forms the basis for syntactic side-effect analysis using the effect system. The results of this section do not deal with effect masking.

**Definition.** For all $\theta$ and $\theta'$ such that $\theta \overset{\text{red}}{\Longrightarrow} \theta'$, let

- $\mathcal{A}(\theta, \theta')$ denote the location(s) allocated in the reduction step $\theta \overset{\text{red}}{\Longrightarrow} \theta'$
- $\mathcal{R}(\theta, \theta')$ denote the location(s) read in the reduction step $\theta \overset{\text{red}}{\Longrightarrow} \theta'$
- $\mathcal{W}(\theta, \theta')$ denote the location(s) written in the reduction step $\theta \overset{\text{red}}{\Longrightarrow} \theta'$

**Claim.** (Effect Soundness) Reduction of a well-formed state allocates, reads, and writes only locations in the regions specified by its effect. In other words, if $\theta \overset{\text{red}}{\Longrightarrow} \theta'$ and $\theta$ ! $\epsilon$ where

$$\epsilon \simeq (\texttt{maxeff}\ (\texttt{alloc}\ \rho_A)\ (\texttt{read}\ \rho_R)\ (\texttt{write}\ \rho_W))$$

then

$$\begin{aligned} \mathcal{A}(\theta, \theta') &\subseteq \rho_A \\ \mathcal{R}(\theta, \theta') &\subseteq \rho_R \\ \mathcal{W}(\theta, \theta') &\subseteq \rho_W \end{aligned}$$

Because reduction preserves or reduces the effect of a state, this claim generalizes immediately to $\theta \overset{\text{red}}{\Longrightarrow}^* \theta'$.

Effect soundness means that the syntactic effect descriptions of the ordinary expressions that constitute a program are a conservative approximation of their actual effects. It follows that this syntactic effect information can be used to identify, at compile time, ordinary expressions that can be memoized and ordinary expressions that can be evaluated concurrently.

## Effect Masking

In the presence of effect masking, the effect soundness property reads as follows:

**Proposition (revised).** (Effect Soundness) Reduction of an expression in a well-formed state allocates, reads, and writes only locations that can be reached through the regions specified by its effect and/or through regions that are accessible only within the expression.

# Index

# OFFICIAL DISTRIBUTION LIST

| | |
|---|---|
| Director<br>Information Processing Techniques Office<br>Defense Advanced Research Projects Agency<br>1400 Wilson Boulevard<br>Arlington, VA 22209 | 2 Copies |
| Office of Naval Research<br>800 North Quincy Street<br>Arlington, VA 22217<br>Attn: Dr. R. Grafton, Code 433 | 2 Copies |
| Director, Code 2627<br>Naval Research Laboratory<br>Washington, DC 20375 | 6 Copies |
| Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22314 | 12 Copies |
| National Science Foundation<br>Office of Computing Activities<br>1800 G. Street, N.W.<br>Washington, DC 20550<br>Attn: Program Director | 2 Copies |
| Dr. E.B. Royce, Code 38<br>Head, Research Department<br>Naval Weapons Center<br>China Lake, CA 93555 | 1 Copy |
| Dr. G. Hooper, USNR<br>NAVDAC-OOH<br>Department of the Navy<br>Washington, DC 20374 | 1 Copy |